Delay Characterization of Cyclone V FPGA

Honors Undergraduate Research Thesis

Presented in partial fulfillment of the requirements for graduation *with honors research distinction* in the undergraduate colleges of the Ohio State University

by

Caitlin Patterson

The Ohio State University

July 2021

Project Advisor: Professor Gregory Lafyatis, Department of Physics

Thesis Committee Members: Professor Daniel Gauthier, Department of Physics; Professor Jim Fowler, Department of Mathematics

Abstract

Each new generation of FPGAs features smaller transistor sizes and more densely arranged features. This thesis updates and builds upon the work of Sedcole and Cheung on the 90 nm Cyclone II FPGA by measuring delay variability in a 28 nm Cyclone V FPGA [P. Sedcole and P. Y. K. Cheung. Within-die delay variability in 90nm fpgas and beyond. In 2006 IEEE International Conference on Field Programmable Technology, pages 97–104, 2006. DOI:10.1109/FPT.2006.270300]. Studying the delay variation is impactful for FPGA applications which capitalize on the inherent delays of logic gates. These include time-to-digital converters and physically unclonable functions. In this thesis, I describe the methodology used to characterize the delays of an entire FPGA chip using ring oscillators made from inverter logic gates. In addition, I describe a method to change the routing of an individual inverter and describe the impact this has on timing delay. I found that for ring oscillators contained wholly within a logic cell, LABs and MLABs on the Cyclone V, there were a range of delays. When configured favorably, LABs had an average delay 226 ps and MLABs had an average delay of 237 ps. Moreover, distributions were bimodal. For example, one peak had an average delay of 225 ps with a standard deviation of 2 ps and the second peak 237 ps with a standard deviation of 3 ps.

Acknowledgments

Thank you to Professors Gregory Lafyatis and Daniel Gauthier for their support and guidance. A very special thanks goes to Peter Menart who is equally responsible for the work documented in this thesis in code, ideas, data, and humor. In addition, I am grateful for the guidance of Noeloikeau Charlot, especially for teaching us how to create location assignments and for contributions of code describing the coordinates on the DE10 Nano FPGA. I would also like to thank Liam Ramsey for contributing very useful code to interface with and read data off of the FPGA and also for his personal support.

Thank you Alex for being adjacent to me while I was writing this document and reminding me to use the geometry package. Special thanks go to comrades Dylan and Harrison, for reciprocity. Finally, thank you Rushil for providing me with no less than 600 mg of caffeine per day but accidentally no more than 1400 kcals.

Table of Contents

Page

List of Figures

Chapter 1: Introduction

In physics, there are a variety of applications for time-to-digital converters (TDCs), which recognizes incoming events and records the time at which they occur. In quantum optics, there is often a need to time-tag individual photons. For example, time-tagging can be used to observe photon anti-bunching, an indicator of a single photon source. Single photon sources are desired in physics for applications in quantum communication, quantum computing, and even quantum sensing [1]. Commercial time-to-digital converters are expensive and contain a fixed number of channels (inputs to the TDC device). However, it is possible to develop time-to-digital converters on low-cost programmable integrated circuits known as field programmable gate arrays (FPGAs) using the intrinsic delays of logic circuits. One advantage of a configurable, replicable TDC design on a low cost circuit is the ability to create a large number of channels. Using a large number of channels can be beneficial in photon counting experiments as described in Ref. [2]. In this thesis, we will explore the logic delays of FPGAs circuits, keeping in mind a specific interest in time-to-digital converter technology.

1.1 FPGA overview

A field programmable gate array (FPGA) is a type of integrated circuit that can be reconfigured for different applications. The circuits are composed of programmable logic

cells and interconnects that can be reconfigured to connect different logic gates. These circuit elements are typically configured in a grid as shown in Figure 1.1.

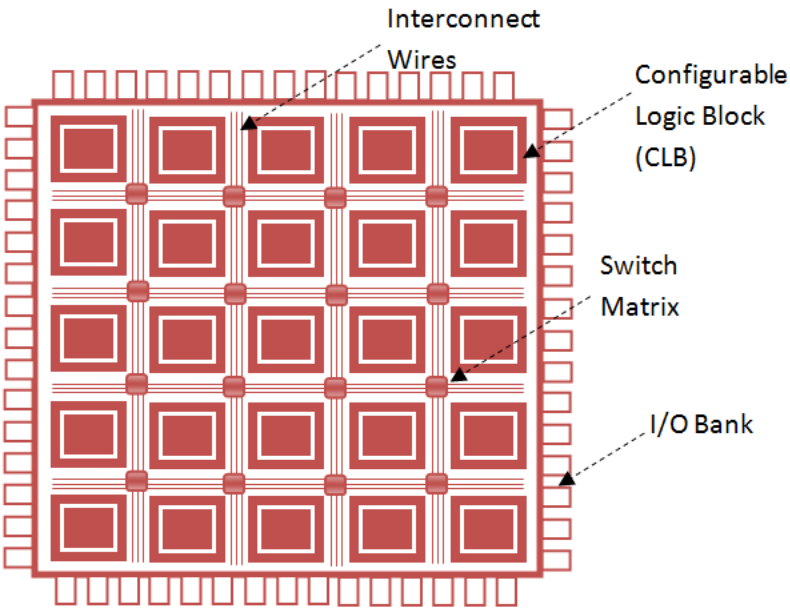


Figure 1.1: General architecture of FPGA chip. Taken from Ref. [5].

For this thesis, I study the Intel Cyclone V FPGA. The programmable logic cells on the Cyclone V FPGA are subdivided into blocks called logic array blocks (LAB) and memory logic array blocks (MLAB). LABs are composed of 10 sub-blocks known as adaptive logic modules (ALM). MLABs also have all of the functionality of LABs, but are also configurable as static random access memory. Each ALM contains six "resources:" two combinational logic cells known as LABCELLs (or MLABCELLs in MLABs) and four register logic cells [3]. In an ALM there is the functionality of two dedicated full adders, an adder carry chain and a register carry chain. This allows the ALM to implement combinational logic, arithmetic functions, and register operations. In each of the two combinitorial resources, there are two 3 inputs LUTs and one 4 input LUT. Overall for combinational logic, each

ALM can implement two 3-input functions or one 6-input function. There is an additional configuration that allows an ALM to implement one function of up to 7 inputs [3]. An image of the chip layout is given in Figure 1.2, with each blue cell representing a LAB or MLAB.

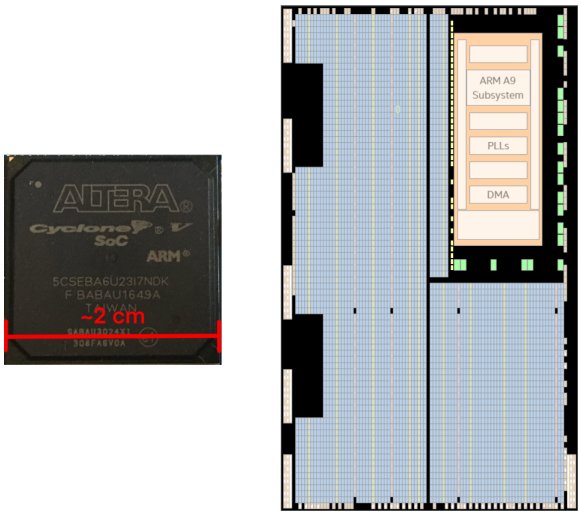


Figure 1.2: The righthand image is a screencap of the Quartus Chip Planner which displays the layout of the FPGA chip. The blue columns are LABs and MLABs, which can be used to implement logic functions. An image of the chip itself given to the left.

The Cyclone V also features a multiple types of interconnect between logic resources. The rows and columns of the chip contain multiple interconnects of varying speed and length. ALMs within the same LAB are connected by fast-local interconnects. Adjacent LABs are connected by direct link interconnects which increase performance by decreasing use of of the row and column interconnects.

FPGA designs are specified in hardware development languages such as Verilog or VHDL. Intel Quartus Prime is the design software used to compile FPGA designs for Intel FPGAs. FPGAs are best for applications that require a specific task, as opposed to general computing purposes. They are well suited to executing parallelizable tasks and are a reusable, low

latency, and flexible platform for developers to implement and prototype circuits on. FPGAs have also becoming increasingly affordable in recent years.

1.2 Asynchronous FPGA Design

Synchronous circuits are digital circuits for which updates to outputs are triggered and synchronized by global clock signals. FPGAs are designed with synchronous circuit design in mind. For example, the input to a register must be stable for a certain amount of time before its output is accessed to avoid metastable states [8]. Metastable states can occur when a flip-flop is accessed before it has stabilized during which its output is unpredictable.

Timing closure refers to the process of modifying a design to meet the timing requirements. In a physical circuit, signal propagation delay in logic gates and wire routing both impact the timing-related circuit performance. To achieve timing closure, circuit designers must manage propagation delay through the circuit and consider the period of the global clocks which govern the design. On the FPGA, a stable clock signal is generated by a crystal oscillator. In a synchronous circuit, the clock period must be large enough so that all flip-flops are stable before the input is latched into the output. In this sense, synchronous circuits have "worst-case" timing performance. When timing closure is achieved in a synchronous circuit, it is protected against timing violations or small variations in the delays of individual logic elements. The logical performance of a synchronous circuit is ideally independent of the individual delays of the logic elements. The *Quartus* compiler assesses the layout of the logic design and may change routing, logic placement, and the implementation of a logical operation when performing timing closure [4].

There are also circuits which include asynchronous logic, or logic not driven by a global clock. In asynchronous designs, output changes are not required to be latched by a global clock signal but instead in response to combinational logic [16]. As FPGAs designs can be

implemented using a large variety of logic and interconnect resources, propagation delays are not very predictable. This can lead to circuit failure due to time difference in arrival of logic inputs or metastable states [16].

However, some asynchronous circuit designers wish to utilize the benefits of the FPGA platform, such affordability, accessibility of development tools, and capability for customization. In particular, we are interested in asynchronous circuits which capitalize on the length of the delays to perform tasks such as timing measurement. For these applications, it is frequently desirable to exploit delays on the FPGA which are low latency and fairly uniform. Interconnect resources are difficult to study, control, and have unpredictable timing delays. On the other hand, the logic gates themselves have small timing delays and are expected to have uniform manufacturing. This makes logic elements such as inverters or carry chains ideal for asynchronous circuit applications which utilize low latency delays.

1.3 Problem Statement: Delay Characterization

To utilize the timing delay of FPGA logic gates, it is necessary to know not only the delay time but also the variation in delay. Over time, FPGA chips have become more complicated and interconnect architectures more intricate [7]. Over time, electronic feature sizes have decreased in an effort to fit more features onto FPGA chips. The Intel Cyclone family of FPGA devices is designed as a low-power, low-cost market product in the FPGA family. Moore's law roughly states that the number of transistors on microchips doubles every year. This is partially due to decreases in transistor size. The Cyclone II FPGA was released in 2004 and featured a 90 nm transistor gate length. The Cyclone V FPGA was released in 2011 and features 28 nm technology.

Decreasing feature size, increased feature density, and increasingly complicated interconnect architecture has lead to greater process variation in FPGAs. This can result in

manufacturing differences between different FPGA chips as well as between the die on the same chip [10]. In addition, these changes have increased the power density of the chip. This results in more heat generation which can impact the various propagation delays of FPGA circuits [11].

As the FPGA is reconfigurable, it is possible to measure the delay of logic gates by measuring the timing of logic circuits on the device. Sedcole and Cheung have measured the delay variation of a Cyclone II 90 nm FPGA using a ring oscillator Built has been performed for the Cyclone II 90 nm FPGA [14]. A ring oscillator is a circuit composed of an odd number of inverters, where the nth inverter is driven by the output of the $(n-1)$th inverter. The 1st inverter is driven by the nth. The nth input is connected to the output of the circuit. Because there is an odd number of inverters in the loop, the circuit never stabilizes and produces a continuously inverting output. This is illustrated in Figure 1.3 that shows a 9 stage ring oscillator as the signal propagates through the circuit.

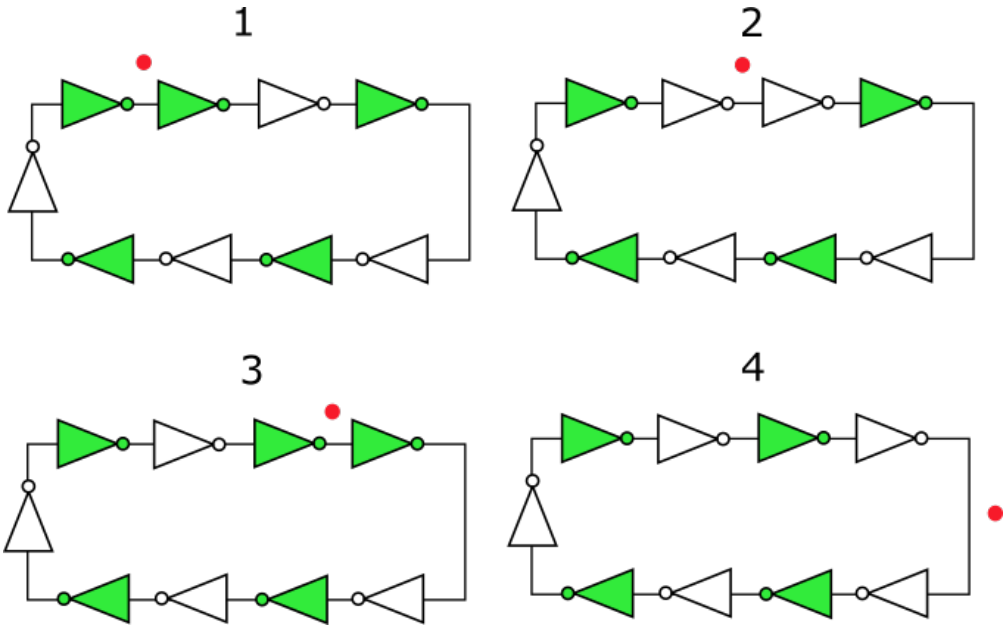


Figure 1.3: A signal propagating a ring oscillator with 9 stages propagating through four consecutive logic elements. This signal is indicated by a red dot.

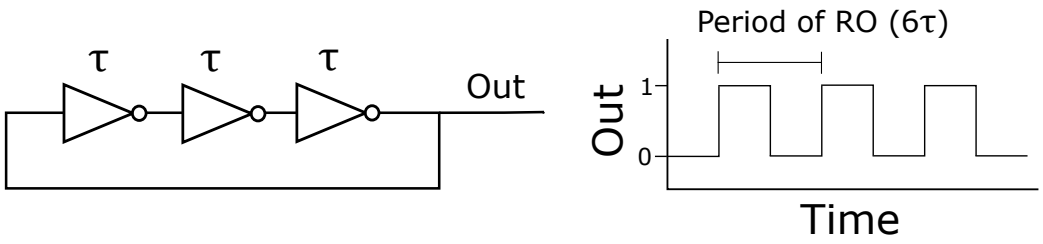


Figure 1.4: Sample ring oscillator and output.

If the inverters are assumed to have a uniform delay of τ, we can expect the output of the ring oscillator to have a period of $2n$. This is because the signal must propagate around the loop twice to achieve a full period. Sedcole and Cheung measured the delay of an inverter in each board position. If T is the period of the ring oscillator, the average delay of the inverters is

$$\tau = \frac{T}{2n}$$

As ring oscillators in their study were placed in single LABs (see Figure 1.5), this assumes that the propagation time between inverters is minimal.

Sedcole and Cheung found they found 3.54% variation in delay per inverter over measurements of multiple 90 nm devices and up to 3.66% variation across a single die [14].

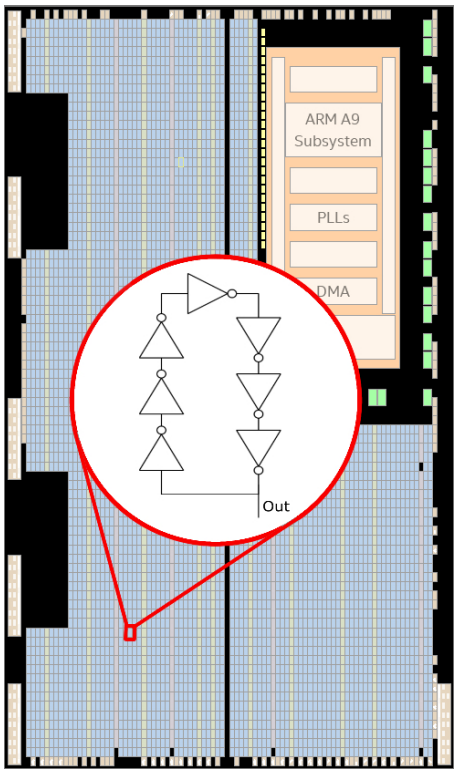


Figure 1.5: Placement of ring oscillator on Chip Planner.

Given the large difference in transistor size and the introduction of the ALM technology to the Cyclone family in the Cyclone V device, the delays of the Cyclone V inverter may be very different from the Cyclone II [3]. For this thesis, a similar ring oscillator delay characterization was performed on a 28 nm Cyclone V FPGA. The specific goals of this study are to:

- Outline a clear method to characterize inverter delay on Intel FPGAs
- Analyze the delays of inverters in each logic cell of a Cyclone V FPGA
- Investigate the relationship between inverter location and delay
- Explore the impact of low-level control of logic implementation in an ALM on inverter delay.

1.4 Applications of Inverter Delay Characterization

Knowledge of the inverter delays is relevant for many circuits which wish to capitalize on FPGA propagation delays. Two very relevant examples are the tapped delay line time-to-digital converter (TDC) and delay-based physically unclonable functions (PUF).

One standard way to implement a TDC in an integrated circuit is to use a tapped delay line. A schematic of a tapped delay line TDC is shown in Figure 1.6. In this circuit, an incoming signal is sequentially sent through many logic elements which serve to delay the incoming signal. Here, the logic elements are inverters. The delay line is tapped after each logic element, with the output of each inverter driving a flip-flop in the design. When a second signal comes into the circuit, all of the flip-flops are consecutively latched. Then, the state of these flip-flops is decoded to indicate how many logic elements the signal had propagated through. If the delay of each logic element is known, the time between the first and second signal can be precisely measured [6]. In the case of Figure 1.6, each inverter has a delay of τ so the TDC would have a resolution of τ.

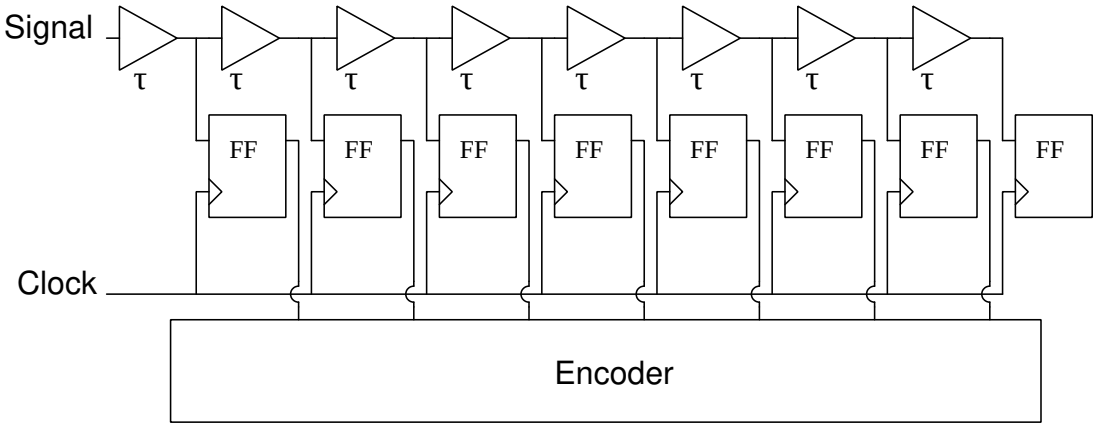


Figure 1.6: Example of a tapped delay line TDC.

In practice, the delay of each inverter would not be uniform. It is possible and necessary to calibrate a tapped delay line TDC to account for this. To optimize performance, it is beneficial to understand the variation in delay between the inverters.

The results in this thesis are also relevant to the implementation of delay-based physically unclonable functions (PUFs). A physically unclonable function is a physical device which returns a “secret" or output when prompted with a “challenge" or input. This secret is extracted from some physical feature of the device that is unique and inherent to it [12]. Many PUFs rely on extracting unique, device-specific information from the physical variations of an integrated circuit which occur in the manufacturing process of semiconductor chips [17]. On FPGAs, the delay time through the routing and transistors varies due to the manufacturing inconsistencies of semiconductor devices heavily packed with features [14]. Therefore, there is interest in implementing PUFs which exploit the delay variations in FPGA circuits.

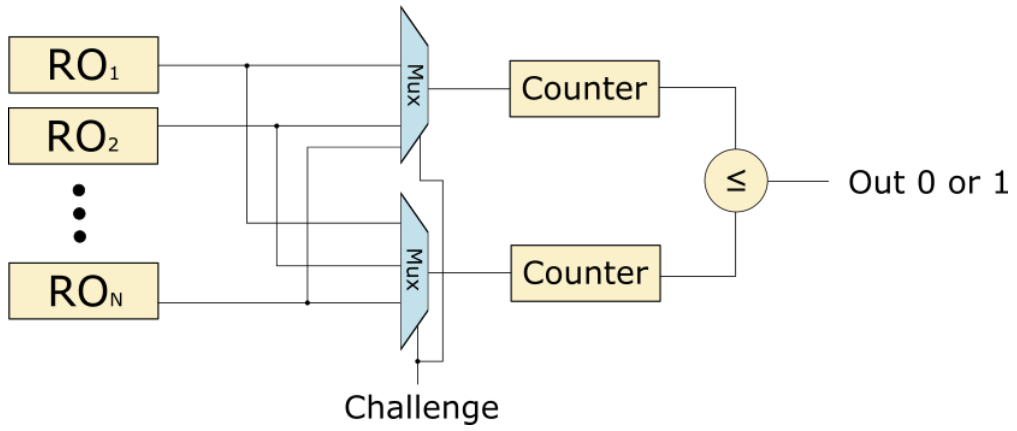


Figure 1.7: Schematic of a ring oscillator based PUF circuit based on the concept from [15].

One delay-based PUF proposed by G.E. Suh and S. Devadas utilizes the inherent variation in the frequency of ring oscillators implemented on an FPGA [15]. A schematic of this PUF is shown in figure 1.7. Their design extracts values from the physical features of the FPGA

from a set of N ring oscillators by comparing their frequencies pairwise. The “challenge" in their physically unclonable function is an input to two multiplexers which selects two ring oscillators from the set. Not every pair should be sampled, as some of the differences in frequency are correlated. Then, the outputs of the chosen ring oscillators are passed to the two counters to measure their frequencies [15].

Depending on which ring oscillator is faster, the PUF outputs a 0 or a 1. This process can be repeated to extract a given number of bits from system. The authors of the study found that implementing identical PUF circuits on two different chips lead to different outputs given the same challenge with a 23% probability. Given identical chips, different outputs occurred with 0.7% probability [15].

In out delay characterization, we will assess some factors that could impact the efficacy of a ring oscillator based PUF. G.E. Suh and S. Devadas of the study noted that because ring oscillator frequencies change as a function of environmental variables, the ring oscillator PUF is vulnerable to errors [15]. This can be accounted for by error correction. However, some errors can be avoided by measuring pairs of ring oscillators whose average frequencies are more dissimilar. This knowledge can be extracted from a delay characterization of the inverters on the FPGA. Moreover, a designer should considering performing a delay characterization of multiple chips when designing the layout of their PUF. For example, if the specific routing design in a region of a type of chip caused lower ring oscillator frequencies across all chips of that design, this might decrease the desired unclonability of the PUF as similar results could be produced on another chip. This is because local and higher level routing between elements of the FPGA is may not be homogenous due to a logic cell's proximity to different types of resources or additional functionality as a memory block.

Chapter 2: Delay Characterization Implementation

In this chapter I discuss how to implement a delay characterization of the Cyclone V 5CSEBA6U23I7NDK FPGA. The code in this section was written in collaboration with Peter Menart and utilizes contributions of Noeloikeau Charlot and Liam Ramsey.

2.1 Verilog Ring Oscillator

In this section, I discuss implementing an inverter based ring oscillator in the Verilog hardware description language. The logic circuit for this ring oscillator is shown in figure 1.4. The Verilog code for the ring oscillator module was adapted from an inverter ring oscillator Verilog design in Davin Rosin's thesis on Autonomous Boolean networks [13].

```
1  module ring_osc (
2          output s_out     //output of RO sent to counter
3          );
4
5  parameter n = 19;       //odd number of inverters
6
7  //make delay line of inverters
8  wire [n-1:0] delay; /* synthesis keep */
9
10 //connect end to the beginning
11 assign delay[0] = ~delay[n-1];
12
13 //connect output to end of oscillator
14 assign s_out = delay[n-1];
15
16 //initialize loop variable
17 genvar i;
18
19 //implement inverters through 0th and 17th element
```

```
20 generate
21         for (i=0; i < (n-1) ; i=i+1)
22         begin: generate_delay
23                 assign delay[i+1] = ~delay[i];
24         end
25 endgenerate
26
27 endmodule
```

In this block of code, there is a parameter n which sets the number of inverters in the ring oscillator. This value must be odd for the circuit to generate an oscillating output. The tilde symbol $\sim$ is a bitwise NOT operation is Verilog. Each bit in the n-bit delay wire is assigned to the inverse of the previous bit. The ring oscillator is then connected in a loop by assigning the first bit in the wire to the inverse of the last wire. Finally, the last bit of in the delay line is assigned to the output of the module. Multiple copies of this module can be instantiated within the larger design to implement as many ring oscillators as needed. A LAB has 20 LABCELLs and can therefore hold 19 inverters.

2.2 Measurement Circuitry

The configurable nature of the FPGA allows the period of a ring oscillator to be measured using logic circuits implemented on the chip itself. We drove a 20-bit counter by the output of the ring oscillator to measure its period, with the counter incrementing on each rising edge of the output. We enabled the counter for one millisecond, thus allowing us to determine the period τ of the ring oscillator using the relation

$$\tau = \frac{1 \text{ ms}}{\#\text{ counts}}.$$

For a ring oscillator with n inverters, the average delay of each inverter is then

$$\tau_{\text{inv}} = \frac{\tau}{2n}$$

This millisecond time interval was achieved by using a phase-locked loop (PLL) to generate a 1 kHz output signal (1 ms clock period). Each stage of the ring oscillator is driven only by

the output of the previous inverter. This is because any enable or disable signal to the ring oscillator could contribute additional delay. Therefore, the ring oscillator has a propagating signal as soon as the board is programmed which continues until the program is desisted.

I conducted two primary measurements in this thesis. In the first, I measured a single ring oscillator repeatedly, to estimate the variation of the delays within a single LAB. In the second, a ring oscillator is placed in each LAB and MLAB of the chip and measured once to study the variation of inverter delay across the chip.

2.2.1 Measurement Circuitry for Repeated Measurements of a Single Ring Oscillator

It is important to allow the ring oscillator to stabilize and the 1 kHz PLL to lock before incrementing the counter. Therefore, we designed the measurement circuitry to wait a fixed period of time before any measurements of the ring oscillator is taken. This is done using a simple clock divider. A counter is driven by some input clock. After the desired number of counts, a register "key_on" is permanently set to the high state. This signal indicates to the control circuitry that enough time has passed since programming to begin the measurement. Moreover, multiple measurements of the single ring oscillator were taken in the same programming. Therefore, a another register "key_start" was used to control the time between measurements. Unlike key_on, key_start oscillated between the high and low states at a fixed frequency with a 50% duty cycle. This is facilitated by a clock divider. When key_on and key_start are both in the high state, the measurement circuitry is enabled and the following process is implemented.

There are three one-bit registers "read," "save," and "endcount" which store the status of the measurement. A 16-bit register RO_reg counts how many measurements have been made of the system. The values of the three registers are assessed and changed at the rising

edge of "clk_count", a 1 kHz clock, to control the measurement as shown below. Note that the initial values of read, save, and endcount are 0.

```
1  if (read) begin
2          read <= 1'b0;
3          save <= 1'b1;
4  end
5  else if (save) begin
6          save <= 1'b0;
7          endcount <= 1'b1;
8          RO_reg = RO_reg + 1;
9  end
10 else if (!key_start && (RO_reg < 600)) begin
11         endcount <= 1'b0;
12 end
13 else if (key_start && key_on && !read && !save && !endcount) begin
14         read <= 1'b1;
15 end
```

As the initial values of all three registers are 0, none of the conditions are satisfied until key_start and key_on are both 1. Once this condition is satisfied, read is set to 1. While read is set to 1, the counter increments at the rising edge of the output of the ring oscillator. On the next rising edge of clk_count, read is set to 0 and save is set to 1. Therefore, the counter is incremented by the ring oscillator during the one millisecond period of clk_count.

While save is set to 1, the value of the counter is pushed to memory and the counter itself is cleared. This is easily achievable before the next rising edge of clk_count. Then, save is set to 0 and endcount is set to 1. At this point, 2 ms have passed as two periods of clk_count have passed. While endcount is 1, we wait until key_start is 0. No conditions are satisfied until this occurs. Therefore, by changing the period of key_start, we are able to roughly manipulate the time between measurements of the ring oscillator. It is important that key_start has a large enough period to accommodate multiple clock cycles of clk_count. Otherwise, the measurement circuitry will not function as desired.

When key_start is turns to 0, endcount is set to 0 on the rising end of clk_count. This returns us to the initial conditions. Once key_start is 1 again, the process will repeat. As

the next measurement begins at the first rising edge of clk_count after key_start is set to 1, the time between measurements is approximately the period of key_start. This is repeated until RO_reg exceeds 600 or the desired number of measurements.

2.2.2 Measurement Circuitry for Single Measurement of Array of Ring Oscillators

In this measurement, the period of each ring oscillator is only measured once. Although it is possible to measure an array of ring oscillators repeatedly, it would be very challenging to push all of the counters to memory in a timely manner before resetting the counters and beginning another measurement. This is because information is latched into memory one clock cycle one clock cycle a time, within the current system. While this challenge could be avoided by measuring the ring oscillators one at a time, we wanted to measure each ring oscillator the same amount of time after the board is programmed.

In this circuit, "key_on" functions as it did for the single ring oscillator measurement by enabling "read" to be set to the high state. Since only one meausurement is performed, only one enable signal is used to initiate the control circuitry. The control circuitry is still driven by the 1 kHz clock, clk_count. The registers read, save, and endcount are initialized to 0.

```
1  if (read) begin
2          read <= 1'b0;
3          save <= 1'b1;
4  end
5  else if (save && (RO_reg >= num_RO)) begin
6          save <= 1'b0;
7          endcount <= 1'b1;
8  end
9  else if (key_on && !read && !save && !endcount) begin
10         read <= 1'b1;
11 end
```

Here, no conditions are satisfied until "key_on" is 1. When this "key_on" is set to 1, the last condition is satisfied and the value of "read" is set to 1. While "read" is set to one, each counter increments at the rising edge of a ring oscillator. At the next rising edge of

"clk_count", the value of "read" is set to 0 and the value of "save" is set to 1. Therefore, "read" is set to 1 for a total of a one millisecond clock period so the counters are incremented for one millisecond. While "save" is 1, the values stored in the counters are moved to memory one by one. Each time a value is moved to memory, "RO_reg" is incremented by one. When "RO_reg" reaches the total number of ring oscillators in the array, no more values are saved to memory. After this, "save" is set to 0 and endcount is set to 1 and the measurement and saving to memory have been completed.

2.3 Location Assignments

To measure a ring oscillator in each LAB and MLAB on the board, we needed to assign each inverter of the ring oscillator to one of the 20 (M)LABCELLs of that (M)LAB. The Intel Quartus Prime software has three necessary compilation steps. The second step is the Fitter (Place and Route) which takes a synthesized design and assigns logical functions to board resources and determines the routing between the logic gates. The Fitter works to find the best logic element location for each design function with the goal of optimizing the use of routing resources and the timing.

User-defined location assignments can also be made, either in the Quartus Chip Planner or by adding assignments to the Quartus Prime Settings File (.qsf). Designs wishing to capitalize on the delay of individual logic elements will often require user defined location assignments. This is because the Quartus compilation software prioritizes resource management and minimizing delays to achieve timing closure in FPGA designs, whereas many autonomous designs such as the tapped delay line TDC or RO PUF incorporate additional delays. The Quartus Prime Settings File contains all entity-level assignments for each individual revision and its syntax is based on the Tcl script syntax.

In this section I will describe how to assign the location of the ring oscillator inverter nodes. In addition, I placed the 20-bit counter driven by each ring oscillator in the (M)LAB above or below. This required setting the location of a 20-bit register in the flip-flops within the (M)LAB as well as the adder for that counter synthesized in the design by Quartus.

The location assignment of nodes can be changed in the Quartus Chip Planner using a "drag and drop" method. However, this must be done manually by the user and therefore is inefficient to execute for larger designs. Instead, it is more convenient to change the location assignments of the nodes using the Quartus Prime Settings File. To do this, lines assigning the location of each node are written into the Quartus Prime Settings File and the Fitter and Assembler portions of the compilation are executed to update the fitting and bitstream.

Here I will show the procedure to assign the location of wires and registers using the Quartus Prime Settings File. Each node in the design has a Full Name and a location. A location assignment for the node can be written in the format

```
1 set_location_assignment <location> -to "<Full Name>"
```

A synthesized wire in the design is placed into a logic cell called a LABCELL or MLABCELL within a LAB or MLAB. Although MLABs have additional functionality, the structure of the two blocks is identical for location assignment purposes. It is necessary to specify if the logic resource is a LABCELL or MLABCELL when referring to the desired location of an inverter.

There are 20 LABCELLs per LAB which are enumerated in multiples of three. So, the nth LABCELL is numbered $3(n-1)$ where $1 \leq n \leq 20$. For example, given X coordinate 1, Y coordinate 1 on chip's grid and position 2 within the LAB, the LABCELL location is called "LABCELL_X1_Y1_N3".

The Full Name of an inverter node can be determined using the design hierarchy or extracted from Quartus. First I will detail how to roughly determine the Full Name using

the design hierarchy. Generally, the Full Name of a node includes the name of the module the wire is defined in, the name of the module instantiation within the highest level module, and the name of the wire. If the module is instantiated within a generate block, the name of the block is also included in the Full Name. If the generate block contains a for loop, the iteration number will also be included. The module name where the wire is defined is listed first, with the name of each lower piece of the design hierarchy listed next in the following format

```
<module name>:<generate block name>[<interation #>].<instantiation name>|<wire name>
```

In cases where the wire is defined within the highest level module, <module name>: can be omitted. Given our ring oscillator code in Section 2.1, the second delay (delay[1]) of a ring oscillator instantiated in the first interation of the generate block (generateRO[0]) would have the following Full Name.

```
ring_osc:generateRO[0].ROinst|delay[1]
```

Therefore, to place an inverter in the ring oscillator with this Full Name in the second LABCELL of the LAB at coordinates X1 and Y1, we must add the following location assignment to the Quartus Settings File.

```
set_location_assignment LABCELL_X1_Y1_N3 -to "ring_osc:generateRO[0].ROinst|delay[1]
    ↪"
```

Given a register within the design, the synthesized node is placed into a flip-flop called an FF located on the LAB. On DE10-Nano Cyclone V FPGA there are 40 flip-flops per LAB or MLAB, and thus 4 flip-flops per ALM. These flip-flops are numbered starting at 1 and ending at 59, with multiples of 3 excluded as these indices are reserved for logic cells. Therefore, a sample register assignment location would look like "FF_X1_Y1_N1".

The Full Name of a register is assigned in the same way as the Full Name of a wire by considering the hierarchy of modules and generate blocks in the *Verilog* code. For example, our program includes a 20-bit counter which is placed below or above the ring oscillator.

Each bit of the counter register is placed is a flip-flop of the ALM. Given one bit of the counter placed above the ring oscillator at coordinates X1, Y1, the first bit of the counter has the following location assignment.

```
1 set_location_assignment FF_X1_Y2_N2 -to counter[0][0]
```

The location of a wire or register within the design hierarchy can be made very complicated. Moreover, the user may be interested in locking down the location of nodes which are created during the synthesis of the project but not defined as wires within a module. Our design utilizes the dedicated adders on each ALM. Therefore, Quartus creates and names each node, and its Full Name must be extracted from the design. To do so, it is convenient to use the Quartus GUI feature **Node Finder**. Node Finder is accessible from the top menu through **View** ▷ **Utility Windows** ▷ **Node Finder**. This will bring up a menu as shown in Figure 2.1.

If specific information about the name of the Node is known, it can be entered into Named to narrow the list of nodes to look through. After this information is selected, or a $*$ is entered to indicate that all nodes should be listed, the List button is used to list the relevant nodes in the project. In Node Finder, the name and location of the individual nodes are given. The Full Name of the desired node can be found by fully expanding the hierarchy of nodes as listed under Nodes Found. The Full Name is written from highest level entity to lowest level entity, with each level of the hierarchy being separated by a vertical bar as previously demonstrated.

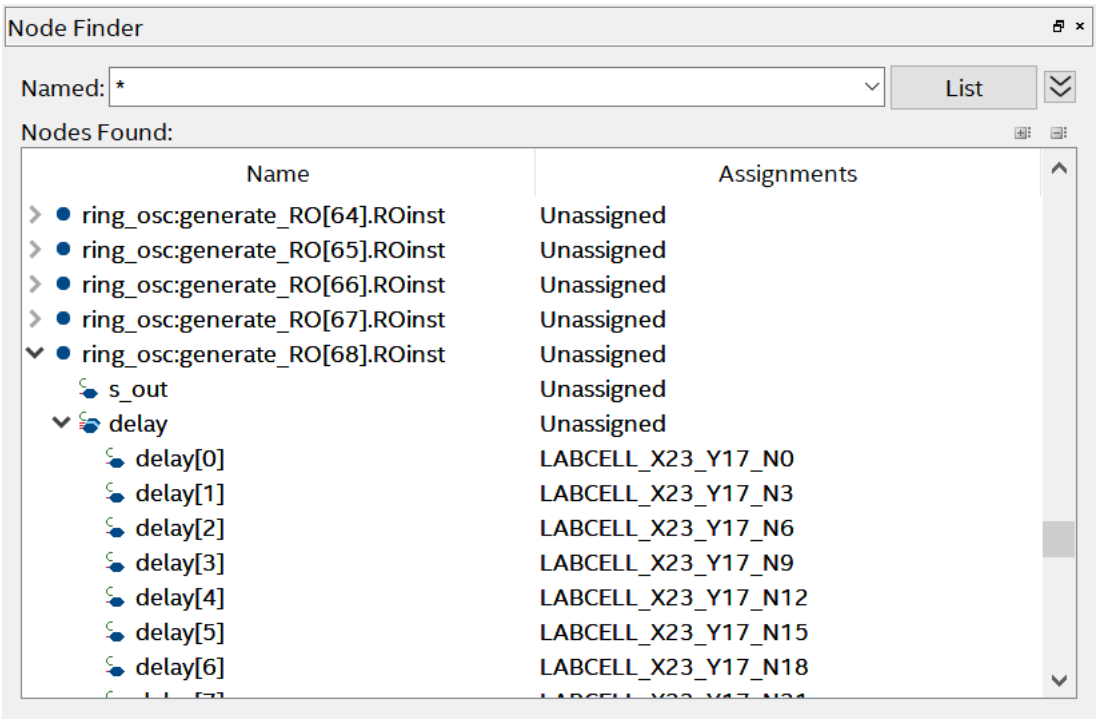


Figure 2.1: Example of Quartus Node Finder

2.4 Location assignments for a full grid of ring oscillators

In this section, I will compare how Quartus synthesizes and places a design with 226 ring oscillators, and compare the placement and delays to a grid of 226 ring oscillators with user-defined location assignments. When Quartus places a ring oscillator with less than 20 inverters, it generally restricts those inverters to the same (M)LAB. However, they are often placed in an arbitrary order down the LAB. Therefore, those ring oscillators may have additional delays between the inverters have delays that are not optimized. We choose to place consecutive inverters in the circuit in consecutive (M)LABCELLs as shown in Figure 2.2. In this figure, the routing interconnect for the (M)LABCELL inputs is also given with blue arrows.

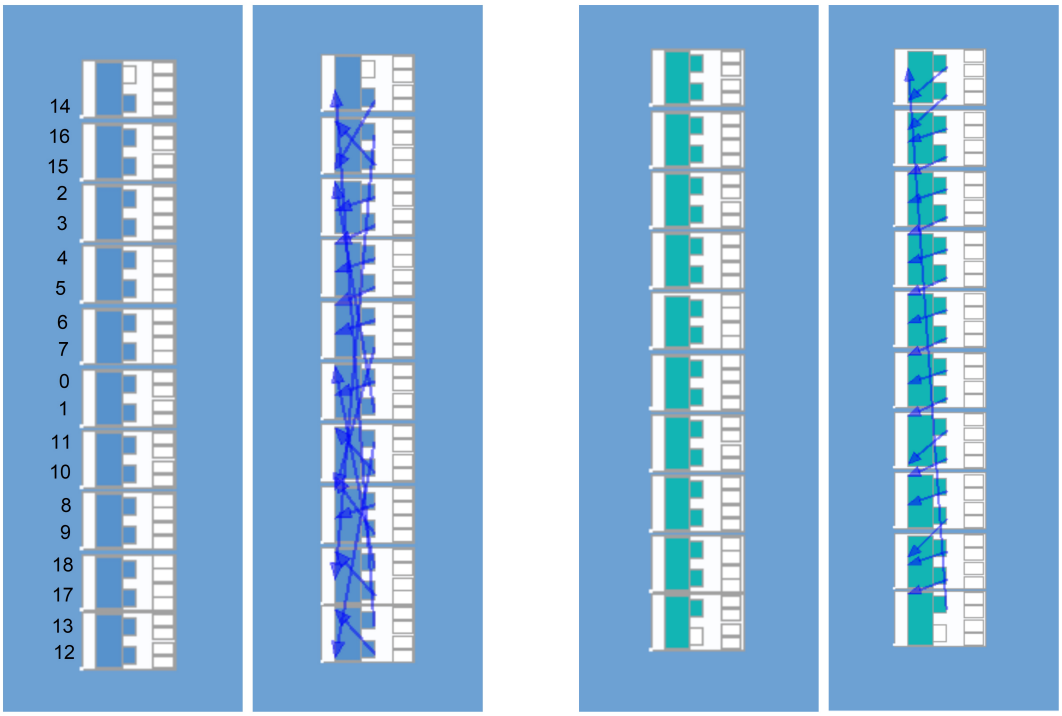


Figure 2.2: Left: a ring oscillator with location assignments designated by the Quartus fitter. Right: with user-defined location assignments for each inverter.

In Figure 2.3, we can get a general idea of how Quartus places multiple ring oscillators on the chip. Most of the design is placed in the bottom portion of the chip. When implementing a grid of ring oscillators, we chose to instead spread them out over the chip to avoid congested routing around each ring oscillator. The average delay per inverter for each configuration of the 226 ring oscillators is shown in Figure 2.4. While the delays are clustered around similar values, the ring oscillators with user-defined location assignments shown a much tighter and more uniform distribution of delay per inverter.

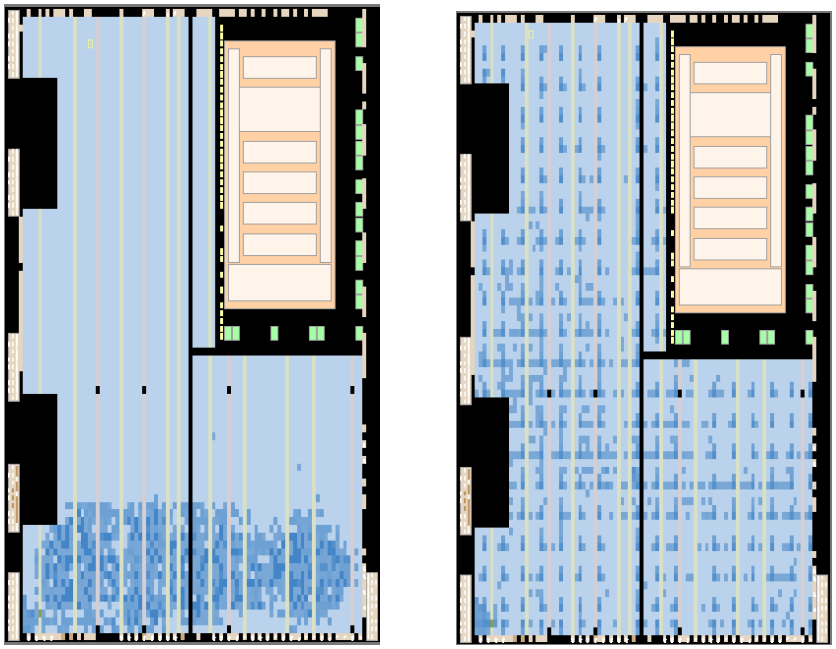


Figure 2.3: 226 ring oscillators in the Quartus *Chip Planner* with user-defined location assignments on the right. The ring oscillators are fixed in a grid with 4 units of horizontal spacing and 3 units of vertical spacing

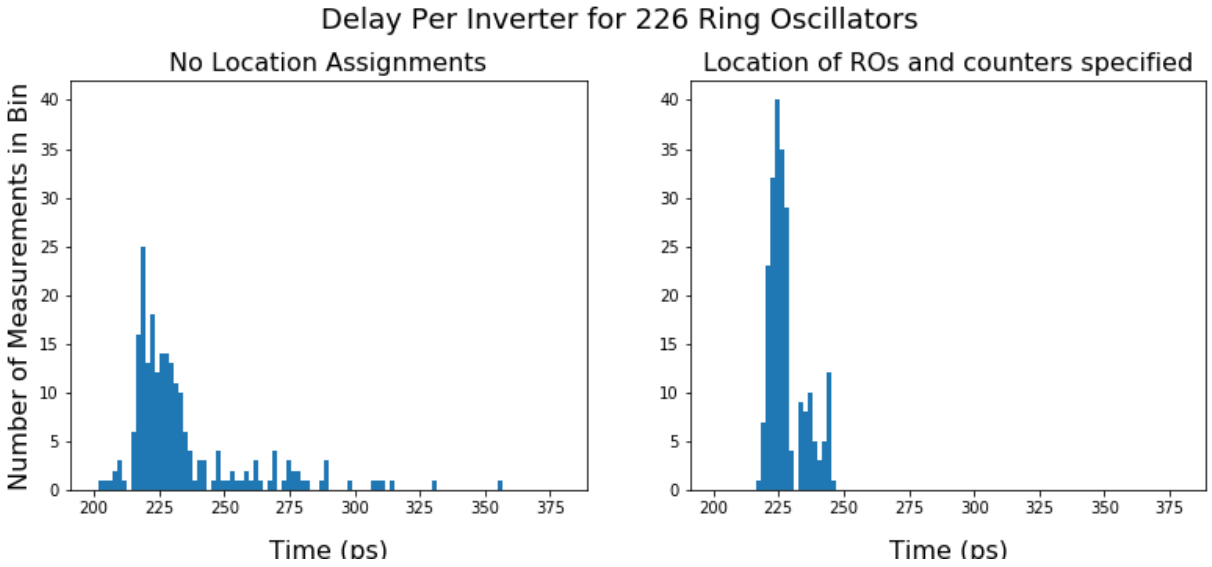


Figure 2.4: Histograms of average delay per inverter for 19 stage ring oscillators with and without user-defined location assignments.

2.5 Specifying Input Routing

In this section I will discuss how to specify the input port to the LABCELL for a specific inverter in a ring oscillator. Due to the configurability of the multiple input ALM, an inverter can be implemented using any of the ALM inputs. For each logic cell, there are six possible inputs including known as DATAA, DATAB, DATAC, DATAD, DATAE, and DATAF. Determining how to implement general changes to the LUT routing of various logic gates can be challenging for multiple reasons, including the unpredictability of the **Quartus** implementation and the detailed knowledge of the circuit needed to implement changes.

In general, it would be very difficult to implement large scale routing changes on complicated designs. The routing fixed by the Quartus compiler is optimized for timing closure and logical accuracy. Therefore, it is only necessary to manually consider the routing when you wish to utilize the delays as part of a design. The most common delays used in digital circuits are inverters and carry chains. A good continuation of this effort would be to refine and assess this technique for carry chains.

The method followed in this section expands on the a tutorial in the book *Hardware Security and Trust* [9]. The ALMs of the Cyclone V differ in detail from the logical elements described in the book but the methods they describe are applicable to the latter. In the book, Sklavos et. al. discuss how using using different inputs to a single input LUT can lead to different timing delays due to the structure of the four-input LUT. They also provide a general overview for changing the LUT input and LUTmask for a single input logic gate in the *Quartus* GUI. After this process, they suggest exporting these changes to a TCL file which can be used to develop a script for applying similar changes across the board. The authors note that the current LUTmasks and input of the logic cell must be known to implement changes to the LUT routing using a TCL script. In this chapter, I will show how

we expanded on the general method given in *Hardware Security and Trust* to implement routing changes for a grid of ring oscillators.

First, we review the method to change the routing of an input in the *Quartus* GUI. This is necessary to understand what pieces of information you need to gather about your LCELL change the LUT routing. It is helpful to first understand the current routing in your LCELLs of interest. Information about the routing of an LCELL can be accessed through the **Resource Property Editor** by double-clicking the LCELL of interest in the Chip Planner. In our design, the ring oscillator inverters would use a variety of inputs. This allowed us to collect preliminary information about the LUTmasks used for an inverter given the input.

To change the LUT routing in the *Quartus* GUI, you only need to know the LUTmask of an inverter for the input you want to use. By viewing various inverters in the Resource Property Editor, we were able to work backwards to determine the LUTmask of an inverter using DATAF, which has . Using this information, we changed the routing of an LCELL with original input DATAC. These changes are also performed in the Resource Property Editor. There are four steps to changing the routing of an inverter in the Resource Property Editor:

(1) Route the signal through the correct input in the **Connectivity** tab
(2) Remove the signal from the old input in the **Connectivity** tab
(3) Update LUTmasks to match inverter LUTmasks for new input

After these steps are complete, the changes can be implemented selecting **Check and Save all Netlist Changes** in the **Change Manager**. Moreover, changes in the **Change Mananger** can be exported to a TCL file by selecting them and selecting Export ▷ Export All Changes As... in the context menu.

After these changes are exported, a TCL script to execute the desired changes is obtained. We can use this script as a template to change the routing for other LCELLs.

To change the routing using a TCL script, the initial state of the LCELL must be known. In the generated TCL script, each change begins with a command **set node_properties** which takes the initial conditions as parameters. These initial conditions must be set before any single change is implemented, with single changes considered to be adding a connection, removing a connecting, or changing one LUTmask. An example is shown below for the first inverter in the first ring oscillator of the array before any changes are made.

```
set node_properties [ node_properties_record #auto \
        -node_name |multRO|ring_osc:generate_RO\[0\].ROinst|delay\[0\] \
        -node_type LCCOMB_SII \
        -op_mode fractured \
        -position top \
        -f0_lut_mask FF00 \
        -f1_lut_mask FF00 \
        -f2_lut_mask FF00 \
        -f3_lut_mask FF00 \
        -fanins [ list \
                [ fanin_record #auto -dst {-port_type DATAD -lit_index 0} -src {
                    ↪-node_name |multRO|ring_osc:generate_RO\[0\].ROinst|delay
                    ↪\[18\] -port_type COMBOUT -lit_index 0} -delay_chain_setting
                    ↪-1 ] \
        ] \
]
```

From this, we can extract that there are nine initial conditions to worry about, node_name, node_type, op_mode, position, f0_lut_mask, f1_lut_mask, f2_lut_mask, f3_lut_mask, and fanins.

The initial condition node_type consistently had the input LLCOMB_SII. The remaining initial conditions depend on which inverter is targeted, specific details of the LCELL, and required the user to take into account changes being implemented in previous stages of the TCL script. We will now go through each of the relevant initial conditions.

- node_name

The input to this is the Full Name. This is the Full Name which is used to find the initial input to the ring oscillator in the routing constraints file.

- op_mode

  The two possible inputs to this are "fractured" and "normal." The input "fractured" indicates that the upper and lower parts of the adaptive logic module (ALM) are used for two different logic functions. The input "normal" indicates that the ALM acts as one unit and performs one logic function. When two inverters are implemented in the ALM, the value of this condition is fractured. When one inverter is implemented in the ALM, the value of this condition is normal. As far as we have found, there is no easy and scriptable way to extract this intiial condition from the design files. Therefore, it is important to be aware of how many inverters are placed in an ALM when writing location assignments and to keep track of this information. When the location of our 19-stage ring oscillator is fixed as in figure 2.2, only the final inverter has an op_mode input of normal. The value of op_mode does not change for a given node.

- position

  The possible of position are "top" and "bottom." This indicates whether the inverter is implemented in the top or bottom half of the ALM. There is no easy or scriptable way to extract whether or not an ALM is in the "top" or "bottom" from the design files. However, we simply needed to remember whether or not a node was placed into the top or bottom of a given ALM. As shown in figure 2.2, our inverters are placed uniformly down the LAB, so it is easy to determine the value of the position input.

- f0_lut_mask, f1_lut_mask, f2_lut_mask, f3_lut_mask

  Before any changes are made, the initial values of the LUTmask depend on the input port being used. For an inverter, the LUTmasks for different inputs are listed in the

Python dictionary Appendix A.2.3. Therefore, when the initial input is extracted from the routing constraints file, it can used to assign these initial conditions. It is important to know that when the ALM is in fractured mode, only two of the LUTmasks can be edited. When the position value is top, only LUT F0 and LUT F2 are configurable. When the position value is bottom, only LUT F1 and LUT F3 are configurable. Moreover, in both cases F0 and F1 have the same LUTmask and F2 and F3 have the same LUTmask. In fractured mode, one only needs to change the two configurable LUTmasks. When the routing is normal, all the LUTmasks are configurable and each must be manually changed.

The LUTmasks must be changed one at a time in the TCL script. Before implementing the next change, it is important to update the initial conditions to reflect that one of the LUTmasks has been changed.

- fanins

  This lists the inputs into the LCELL using their FUll Name, as well as the input port they use. In our design, the input into an LCELL is the output of the previous inverter. So, the node_name within the fanins input will be the Full Name of the previous inverter. It is important to note that all the fanins must be listed in the node properties. Therefore, after the new input connection is added, but before the old input connection is removed, both fanin connections must be accounted for. This can be seen in the template for adding a connection to the LCELL in Appendix A.2.3.

A general node_properties assignment is shown below, now with node specific information removed. Note that this assignment only has one fanin, so it does not represent the node_properties for all assignments.

```
1
2 set node_properties [ node_properties_record #auto \
```

```
        -node_name |multRO|ring_osc:generate_RO\[<INSERT_RO_INDEX>\].ROinst|delay\[<
            ↪INSERT_CURRENT_DELAY>\] \
        -node_type LCCOMB_SII \
        -op_mode <INSERT_MODE> \
        -position <INSERT_POSITION> \
        -f0_lut_mask <INSERT_LUT0> \
        -f1_lut_mask <INSERT_LUT1> \
        -f2_lut_mask <INSERT_LUT2> \
        -f3_lut_mask <INSERT_LUT3> \
        -fanins [ list \
                [ fanin_record #auto -dst {-port_type <INSERT_INPUT> -lit_index 0}
                    ↪-src {-node_name |multRO|ring_osc:generate_RO\[<
                    ↪INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_PRIOR_DELAY>\]
                    ↪-port_type COMBOUT -lit_index 0} -delay_chain_setting -1 ] \
        ] \
]
```

2.5.1 Adding a Connection to the LCELL

The full template script to add the connection for the new input is given in Appendix A.2.3. Now I will discuss the specific tcl commands required to add a connection to the node. In this case, we are taking the signal that is sent into the original input to the LCELL, and also sending it to the desired input. This requires the specific tcl command

```
set result [ make_ape_connection_wrapper $node_properties |multRO|ring_osc:
    ↪generate_RO\[<INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_CURRENT_DELAY>\] <
    ↪INSERT_NEW_INPUT> 0 |multRO|ring_osc:generate_RO\[<INSERT_RO_INDEX>\].ROinst|
    ↪delay\[<INSERT_PRIOR_DELAY>\] COMBOUT 0 -1 ]
```

Here <INSERT_RO_INDEX> should be replaced with the index of the ring oscillator being considered and <INSERT_CURRENT_DELAY> should be replaced with the index of the inverter being considered. The value <INSERT_NEW_INPUT> is replaced with the new desired input, i.e. DATAF or DATAD. The value <INSERT_PRIOR_DELAY> refers to the index of the previous inverter in the ring oscillator. In the node_properties for this change, there is only one fanin corresponding to the original input.

2.5.2 Removing a Connection from the LCELL

The full template of a TCL script for removing the connection to the old input is given in Appendix A.2.3. After the new connection is added to the LCELL, the old input port must be disconnected. As a new connection has just be added, there are two fanins to the LCELL, both of which come from the same node. Therefore, the input for the fanins property is

```
1 -fanins [ list \
2         [ fanin_record #auto -dst {-port_type <INSERT_OLD_INPUT> -lit_index 0} -src
             ↪{-node_name |multRO|ring_osc:generate_RO\[<INSERT_RO_INDEX>\].ROinst|
             ↪delay\[<INSERT_PRIOR_DELAY>\] -port_type COMBOUT -lit_index 0} -
             ↪delay_chain_setting -1 ] \
3         [ fanin_record #auto -dst {-port_type <INSERT_NEW_INPUT> -lit_index 0} -src
             ↪{-node_name |multRO|ring_osc:generate_RO\[<INSERT_RO_INDEX>\].ROinst|
             ↪delay\[<INSERT_PRIOR_DELAY>\] -port_type COMBOUT -lit_index 0} -
             ↪delay_chain_setting -1 ] \
4 ]
```

To remove the connection, the following TCL command is used.

```
1 set result [ remove_ape_connection_wrapper $node_properties |multRO|ring_osc:
    ↪generate_RO\[<INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_CURRENT_DELAY>\] <
    ↪INSERT_OLD_INPUT> 0  ]
```

The values of <INSERT_RO_INDEX>, <INSERT_CURRENT_DELAY> and <INSERT_OLD_INPUT> are as previously discussed.

2.5.3 Changing a LUTmask

The full template of a TCL script for changing a LUTmask is given in Appendix A.2.3. After the new input is connected to the LCELL and the old input is disconnected from the LCELL, the there is only one node in the fanins list. Therefore, the input for the fanins property is

```
1 [ fanin_record #auto -dst {-port_type <INSERT_NEW_INPUT> -lit_index 0} -src {-
    ↪node_name |multRO|ring_osc:generate_RO\[<INSERT_RO_INDEX>\].ROinst|delay\[<
    ↪INSERT_PRIOR_DELAY>\] -port_type COMBOUT -lit_index 0} -delay_chain_setting -1
    ↪ ]
```

To change the LUTmask the following TCL command is used.

```
set result [ set_lutmask_wrapper $node_properties |multRO|ring_osc:generate_RO\[<
    ↪INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_CURRENT_DELAY>\] "F<INDEX_LUTMASK>
    ↪LUT Mask" <NEW_LUTMASK> ]
```

The values of <INSERT_RO_INDEX> and <INSERT_CURRENT_DELAY> are as previously discussed. The value of <INDEX_LUTMASK> is 0,1,2,3 depending on which LUTmask is being configured. The value of <NEW_LUTMASK> is the length four string code corresponding to the LUTMASK. For an LCELL in fractured mode, only two of the LUTs are accessible. Therefore, only two changes must be made overall, and we can assume that changes to either one of f0_lut_mask or f1_lut_mask will affect the other. The same is true for f2 and f3. However, for an LCELL in normal mode, all four LUTs are accessible and therefore each LUTmask must be changed individually.

To scale up these changes, the original input to each LCELL must be known. This can be found by writing routing changes to a file called the routing constraints file and then executing many of these TCL scripts. This process will be described in the following section.

2.6 Python Scripts

The delay characterization was implemented using 20 different programming files. This required thousands of location assignments, routing changes, and many different data collections. Therefore, we wrote two Python programs: the first to script the compilation of the FPGA bitstreams and the second to program the FPGA with each script. For each compilation file, the script chooses a grid of logic cells to place ring oscillators in. This is described by the function **generate_grid** in Appendix A.2.2. The script separates the grid into two subgrids, identifying coordinates of LABs and MLABs. The Verilog script **multRO** is then altered to instantiate the correct number of ring oscillators in the grid using the function **write_verilog** in Appendix A.2.2. The script then writes the location assignment code for the ring oscillator for each grid point and puts a counter in the grid point above or below in

place_lab and **place_mlab**. These lines are written into the function in **write_qsf**. The code can also generate a reference file which corresponds the index of each ring oscillator to the LAB or MLAB it was placed in in the function **write_ref**. Finally, the function **compile_quartus** compiles the project and produces an ".sof" compilation file.

Additional Python functions are needed to change the routing of the project. To determine the LCELL routing for each inverter, we must parse a file called the routing constraints file. This file contains lines of code which indicate the input port for each inverter. To produce this file, we execute the following tcl command.

```
quartus_cdb <project name> --back_annotate = routing.
```

The Python script parses the routing constraints file to find lines of the form

```
dest = ( ring_osc:generate_RO[<index_RO>].ROinst|delay[<index_delay>], DATAA ),
    ↪route_port = <original input>;
```

to scrape out <index_RO>, <index_delay> and <original_input> from the routing constraints file and put them in a dictionary where you can use (index_RO,index_delay) to obtain <original_input>

A dictionary of the original inputs is then created in the function **make_rcf_dict**. This function is described in Appendix A.2.3. Any inputs that deviated from the desired inputs are then changed using the templates described in section 2.5. The Python functions replacing the inputs to these templates are described in the functions **addremove_connection** and **change_lutmask** in Appendix A.2.3. Between compilations, it is **essential** that the .qsf file and .rcf file are deleted. Otherwise, Quartus will continue to reference the information contained in these files.

Chapter 3: Results

The measurements in this section were performed on the Cyclone V 5CSEBA6U23I7NDK FPGA. In this section, I will discuss various measurements made of ring oscillators on the FPGA. The majority of the measurements were done on ring oscillators with 19 stages measured for one millisecond. Let the delay per inverter be τ, the number of counts be C and the time of measurement be t_{meas}. Then, τ is related to C by

$$\tau = \frac{t_{\text{meas}}}{2 \times \#\text{stages} \times C}.$$

The average delay of an inverter is around 220 ps. Therefore, over a millisecond, we expect the counter connected to the output of the ring oscillator to count 120,000 periods. It is possible that the counter misses a stage when being enabled or disabled. When the ring oscillator is measured for sufficiently long, the impact of this error is miniscule. Given our number of stages, our measurement time, and 120000 counts, a two count error results in a delay per inverter of 219.295-219.302 picoseconds. This is an error of less than 0.01%.

To ensure that the PLL used to dictate the measurement period had locked and the ring oscillator had stabilized, we waited at least 0.5 seconds before taking any measurements of the ring oscillators, but commonly more.

First, we measured a single 19-stage ring oscillator repeatedly over time to characterize the variation in delay of a single ring oscillator. Then, we measured a 19-stage ring oscillator placed in each LAB and MLAB on the board. We measured the period of each ring oscillator for one millisecond beginning ten seconds after initialization. This measurement was

performed on two different FPGAs. After this, we specified the routing through the LUT in each ring oscillator and repeated our full board measurement. Finally, we considered how the delay per inverter is related to the number of stages in the ring oscillator.

3.1 One RO measurements

First, we measured of a ring oscillator placed in a single LAB on the board to understand the variation in the delay of a single ring oscillator. When repeated measurements were made, there were a very small number of outliers with an extremely short delay. This occurrence was repeatable, and may point to some problem with the measurement circuitry for this implementation. These outliers were excluded from the following analysis. The results of this measurement are seen in Figure 3.1.

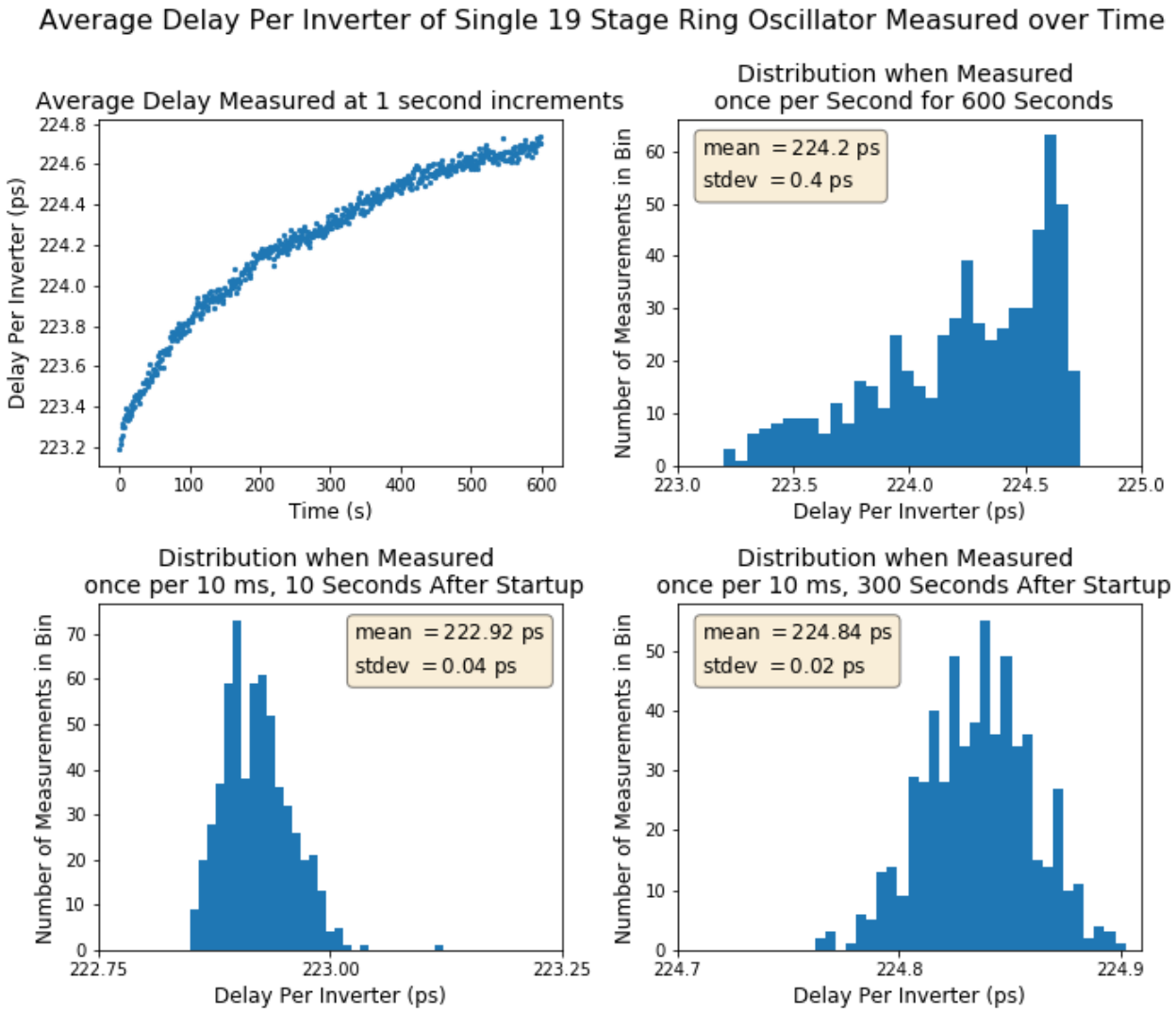


Figure 3.1: The delay per inverter of a single 19-stage ring oscillator measured over time. The top left figure shows the delay per inverter over time as the ring oscillator continuously runs. The top right figure shows the distribution of the measurements in the left plot. The bottom plots show the delay per inverter of a ring oscillator measured more frequently (once per 10 ms) 10 seconds after initialization and 5 minutes after initialization.

The delay per inverter in the ring oscillator was around at around 223 ps right after being initialized and eventually rose to around 225 ps by the end of the 10 minute measurement. This increase happened steadily over time. The rate of increase of the delay also seemed to be decreasing with time. Because of this, there are more values close to 225 ps than 223 ps.

This is reflected in the distribution of the delays. The ring oscillator had an average period of 224.2 ps with a standard deviation of 0.4 ps. This is a variation of around 0.2%. In addition, we measured a ring oscillator 600 times 10 seconds after its initialization and 300 second after its initialization. With a startup time of 10 s, we measured the ring oscillator to have an average delay of 222.92 ps with a standard deviation of 0.04 ps. This measurement had a relative standard deviation 0.02% Surprisingly, this value is lower than the smallest measured delay when the ring oscillator was run for 10 minutes. This could be due to environmental conditions or natural variation. We measured the ring oscillator to have an average delay of 224.82 ps with a standard deviation of 0.02 ps after oscillating continuously for five minutes. This is a relative standard deviation of around 0.01%.

Even when we measured the ring oscillator over 10 minutes, the overall variation in delay per inverter was quite small, with a relative standard deviation of 0.4%. The relative standard deviation is even smaller when measuring the ring oscillator for a shorter period of time. While the variation was small, the delay per inverter increased steadily over time. This indicates that our measurements in the following sections are very depending on how long a ring oscillator has been running, and that each ring oscillator should be measured after running for an equal amount of time.

3.2 Higher Harmonic Oscillators

First, we measured the period of a 19-inverter ring oscillators placed in each LAB and MLAB of a DE10 Nano Cyclone V FPGA. The full results of this initial measurement are given in Figure 3.2, a histogram of the periods of the ring oscillators. The majority of the ring oscillators had a measured period between 8 and 9.5 nanoseconds. However, ring oscillators in 15 of the LAB/MLABs (0.4% of total number), had a measured period of around 3 nanoseconds.

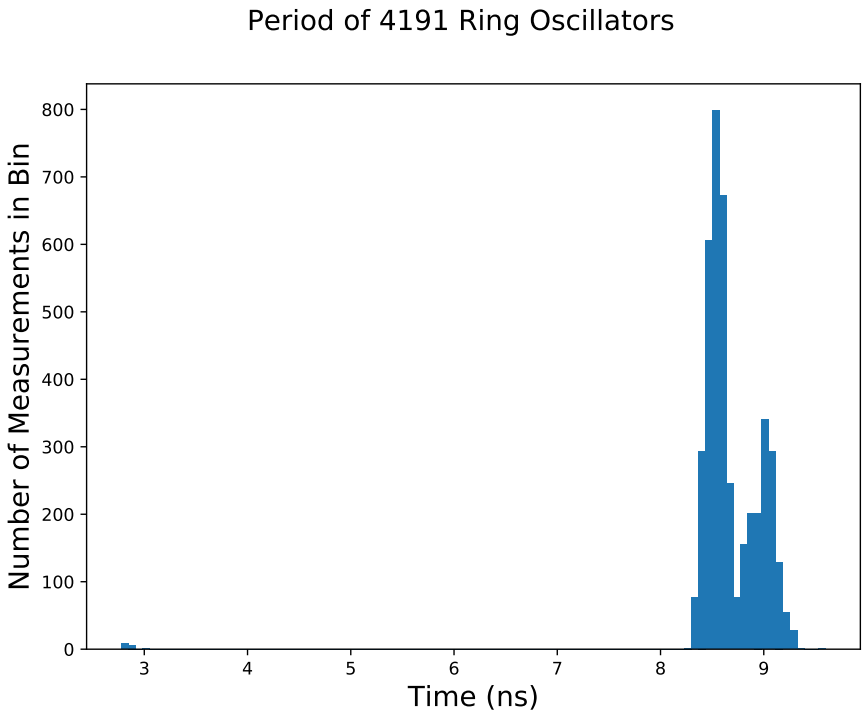


Figure 3.2: The distribution of periods for 19 stage ring oscillators placed in each LAB and MLAB of the FPGA. The majority of the ring oscillators have a period of 8 and 9.5 ns, while a small number have a measured period of 3 ns

A period of 3 ns corresponds to an average delay per inverter of around 75 ps. This is inconsistent with previous measurements of the propagation delay of an inverter [13]. Notably, all outliers were approximately three times as fast as the propagation delay of the bulk of the measurement. While we expect one signal to propagate through the ring oscillator as shown in Figure 1.3, it is also possible for more than one signal to propagate through the ring oscillator. This is illustrated in Figure 3.3, which shows a 19 stage ring oscillator with three signals propagating through it.

For every period of the ring oscillator with one propagating signal, the third harmonic ring oscillator would increment three counts. Therefore a third harmonic ring oscillator would appear to have one third of the expected period. To assess if this is a reasonable

explanation, we implemented a ring oscillator with one NOT gate replaced by a NAND gate. When To assess if this is a reasonable explanation, we implemented a ring oscillator, but with one inverter replaced by a NAND gate. The logic circuit for this NAND ring oscillator is shown in Figure 3.4. Referring to the NAND truth table in Figure 3.5, we see that when Enable is 0, Out should always be 1. When Enable is 1, the NAND gate acts as an inverter with respect to the value of Out, and so the logic circuit behaves as a ring oscillator with 19 stages. In theory, if the system is initalized with Enable = 0, the outputs of the inverters will stabilize so that Out = 1. Then, when Enable is switched to 1, one signal will begin to propagate around the ring oscillator as desired. We repeated our measurement of each LAB and MLAB on the board using NAND ring oscillators. Across multiple trials, we observed no outliers. This supports the hypothesis that the outliers with 3 ns periods were in the third harmonic.

It was initially surprising that the next smallest number of propagating signals is three. However, it is only possible for an odd number of signals to propagate through the ring oscillator. This can be seen by assuming that there is no consecutive pair of inverters with the same output between two propagating signals. If at a given slice in time, one signal correponds to two inverters outputting 0 and the other corresponds to two consecutive inverters outputting 1, there must be an even number of inverters between them. If both output the same value, there must be an odd number of inverters between them. So, two propagating signals, are connected by an odd number of inverters on both sides or an even number of inverters on both sides. As we are considering each signal within the context of two inverters, and assuming that there are an even additional number of inverters in the system, two signals propagating through the system would imply that there is an even total number of inverters in the ring oscillator. We conclude that two signals cannot propagate

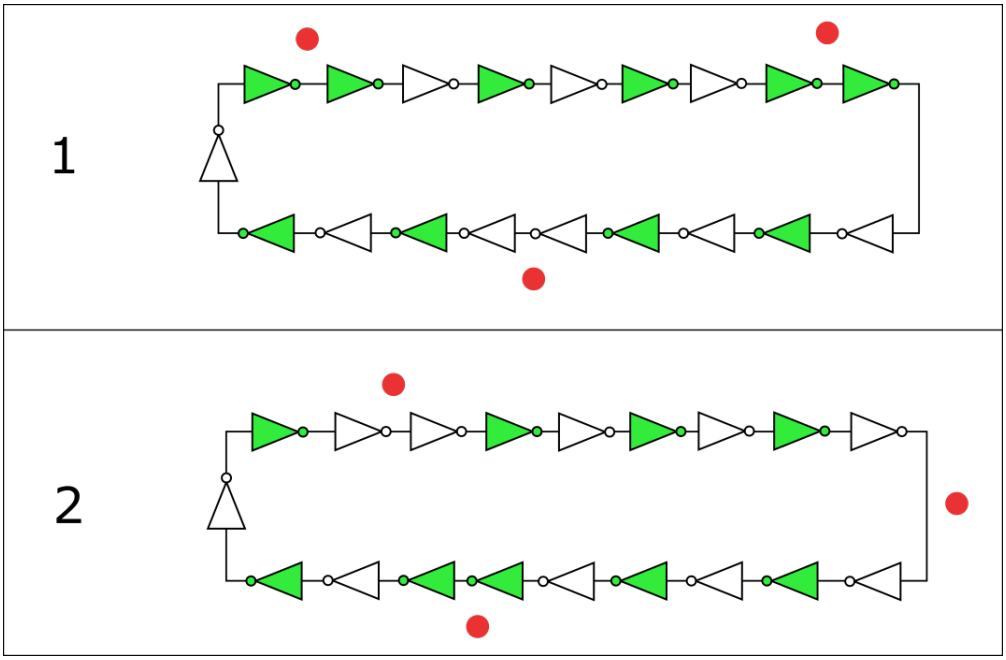


Figure 3.3: Three signals propagating through a 19 stage ring oscillators. The three red dots indicate the location of each signal. Frames 1 and 2 are taken consecutively, as the signal moves from one inverter to the next. In this image, the green inverters have an output of 1 while the white inverters have an output of 0.

through the ring oscillator. This logic can be generalized to any even number of signals. The second lowest possible harmonic is the third harmonic.

Of course, the third harmonic measurements do not represent the actual propagation time of each signal through the ring oscillator. While we could include the third harmonic values in our data by multiplying their periods by three, we choose to exclude them as it is unclear how the multiple signals impacts the actual period of the ring oscillator or the thermalization of the logic elements. This did not have a serious impact on our data analysis third harmonic ring oscillators generally accounted for less than 1% of of the ring oscillators on the board.

Any ring oscillator on the board is susceptible to a third harmonic. However, it is unlikely that a stable ring oscillator with one propagating signal will move to a higher harmonic,

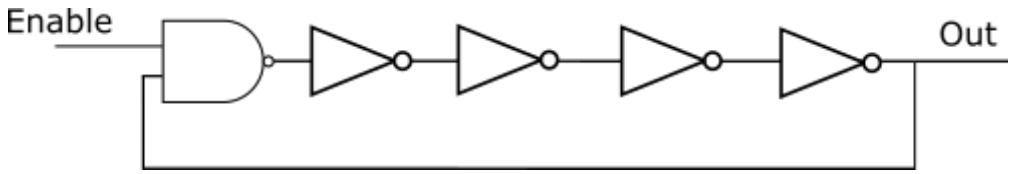
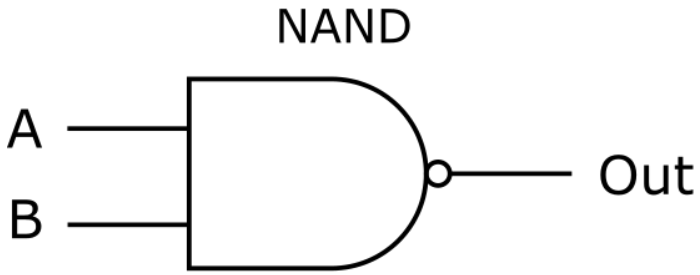


Figure 3.4: A ring oscillator with one NOT gate replaced by a NAND gate for an even number of NOT gates and one NAND gate. When Enable is 0, the NAND gate always outputs 1, and when Enable is 1, the NAND gate inverts the output of the last inverter in the chain. See NAND truth table in Figure 3.5



Input A	Input B	Out
0	0	1
0	1	1
1	0	1
1	1	0

Figure 3.5: NAND truth table. When Input A is 0, Out is always 1. When Input A is 1, the NAND gate acts as an inverter with respect to Input B

and therefore we expect that these measurements are a result of some startup conditions. Anecdotally, some LABs and MLABs seemed to be prone to third harmonic ring oscillators.

While it is unlikely that ring oscillators in a design will end up in a third harmonic, it is especially important to be aware of the possibility when a large number of ring oscillators are included in a design. Additionally, implementing ring oscillators with a larger number of inverters could increase the probability of encountering higher harmonics. I am curious

how nonhomogenous routing between inverters - a feature which might be inherent to the *Quartus* Fitter implementation - impacts the stability and occurrence of higher harmonic behavior. I expect, however, that this would be an extremely challenging thing to assess and is beyond the scope of the work done in this thesis.

If it is imperative to avoid third harmonics, the designer can consider implementing the NAND ring oscillator shown in Figure 3.4. In that case, it is important to note that the NAND gate may have a different delay than the NOT gate. In a large ring oscillator which is prone to higher harmonics, the small variation caused by the different NAND gate delay would be less significant than it is in a smaller ring oscillator.

3.3 Delay Characterization of Cyclone V with 19-Stage Ring Oscillators

In this section, we discuss the delay characterization of a Cyclone V FPGA. A ring oscillator was placed in each LAB and MLAB of the FPGA board as described in Chapter 2. The ring oscillators were run for ten seconds before measurement began and the measured for one millisecond.

We measure the period of a 19-inverter ring oscillator placed in each LAB and MLAB of a Cyclone V chip to understand how the delay of an inverter varies throughout the LABs and MLABs. The higher harmonic measurements discussed in 1.2 are excluded from this analysis. The results of this measurement are shown in the histogram and the heatmap of delay values at their on chip measurement coordinates in Figure 3.6. We see that the distribution of delays is bimodal with two distinct peaks, one roughly centered at 225 ps and the other roughly centered at 237 ps when the distribution is divided at 230 ps. The first peak has a standard deviation of 2 ps or around 0.9% and the second peak has a wider spread with a standard deviation of 3 ps or around 1.3%. In comparison, the delay of an inverter in a single ring oscillator measured repeatedly ten seconds after initialization had

a standard deviation of 0.04 ps or around 0.02%. Even when measured over a long period of time, the relative standard deviation of a the delay per inverter of a ring oscillator in the same LAB is smaller than the standard deviation of either peak, with a value of 0.4 ps or 0.2%. Around 65% (2724 measurements) of the data is in the first peak and around 35% (1452 measurements) of the data is in the second peak.

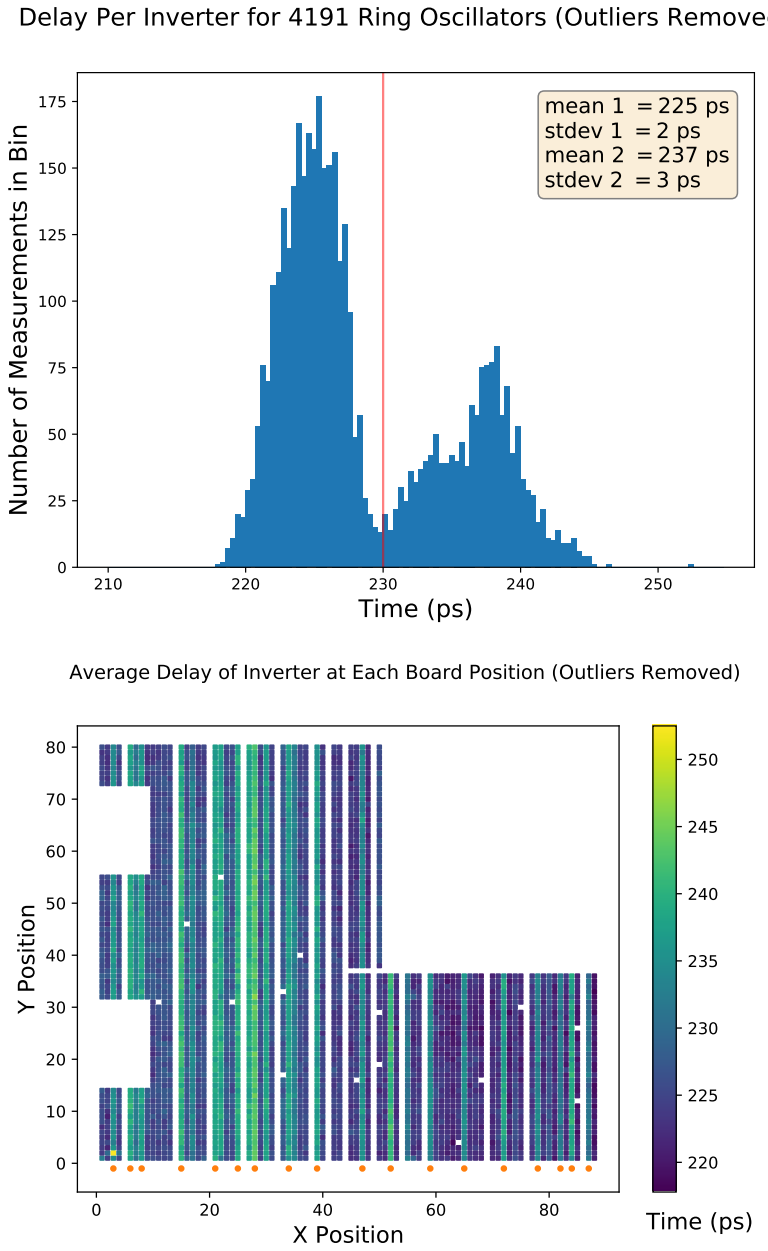


Figure 3.6: A histogram and heatmap of the average delay per inverter for 19 stage ring oscillators in each LAB/MLAB on the board. This distribution is bimodal, with one peak centered at around 225 ps and another centered at around 237 ps. The statistics for the two different modes were computed by roughly dividing the data into two groups, delay less than 230 ps and delay greater than 230 ps. The red line on the figure indicates this divide. This colorplot shows the average delay of an inverter given the position of the LAB/MLAB on the board (Recall Figure 1.2). An orange dot below the column indicates that it is a column of MLABs. Third harmonic outliers are removed.

In the heatmap which shows the coordinate of each measurement, we can observe that the variation in the delay within the columns is quite small. This correlation is expressed in the stacked histogram in Figure 3.7, where we see that the MLABs are mostly located in the second peak. There are still LABs in the second peak. The LABs have an average delay of 226 ps per inverter while the MLABs have an average delay of 237 ps per inverter. Additionally, Figure 3.8 shows the heatmap of each peak. This again reinforces the homogeneity of delays within the column and shows that the second peak is majority MLABs.

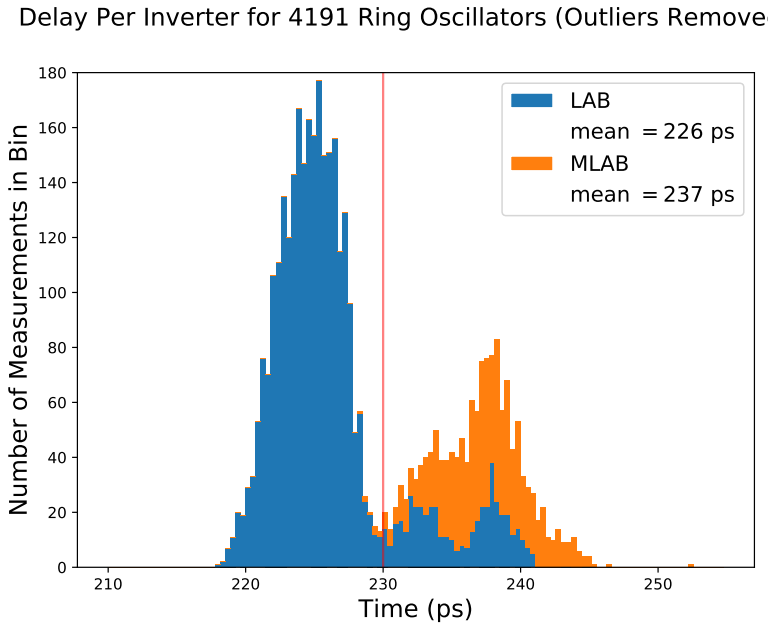


Figure 3.7: A stacked histogram of the delays per inverter, with the blue representing measurements from ring oscillators in LABs and the orange representing measurements from ring oscillators in MLABs

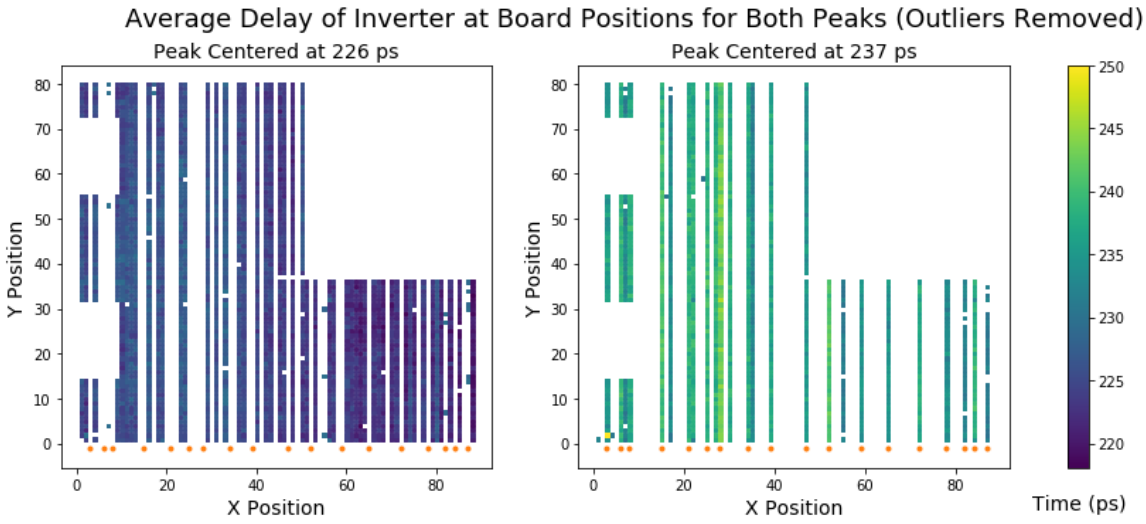


Figure 3.8: Heatmaps of the delay per inverter given the coordinate of the LAB/MLAB. This is separated into two plots depending based on whether the delay is less than or greater than 230 ps each peak. The orange dots denote the columns of MLABs.

.

3.4 Measurement of Different Boards

In addition, this characterization code was implemented on two different DE10 Nano Cyclone V boards. The results of this are shown in Figure 3.9. The majority of the results in this thesis were measured on Board 1. The majority of the LABs/MLABs on Board 1 have an average inverter delay of between 215 ps and 250 ps with an overall average delay of 229 ps. The average inverter delay on Board 2 ranges between 195 ps and 230 ps with an overall average delay of 211 ps. Therefore, the propagation delay through an inverter on Board 2 appears to be somewhat faster than Board 1, but not significantly so compared to the delays for each chip. Moreover, the distribution of delays for Board 2 does still appear to have two peaks, but they are far less distinguishable than for Board 1. A heatmap representation of the delay at each coordinate for both boards is give in Figure 3.10. It appears that similar columns of LABs/MLABs have higher delays between the two boards relative to the rest of the columns.

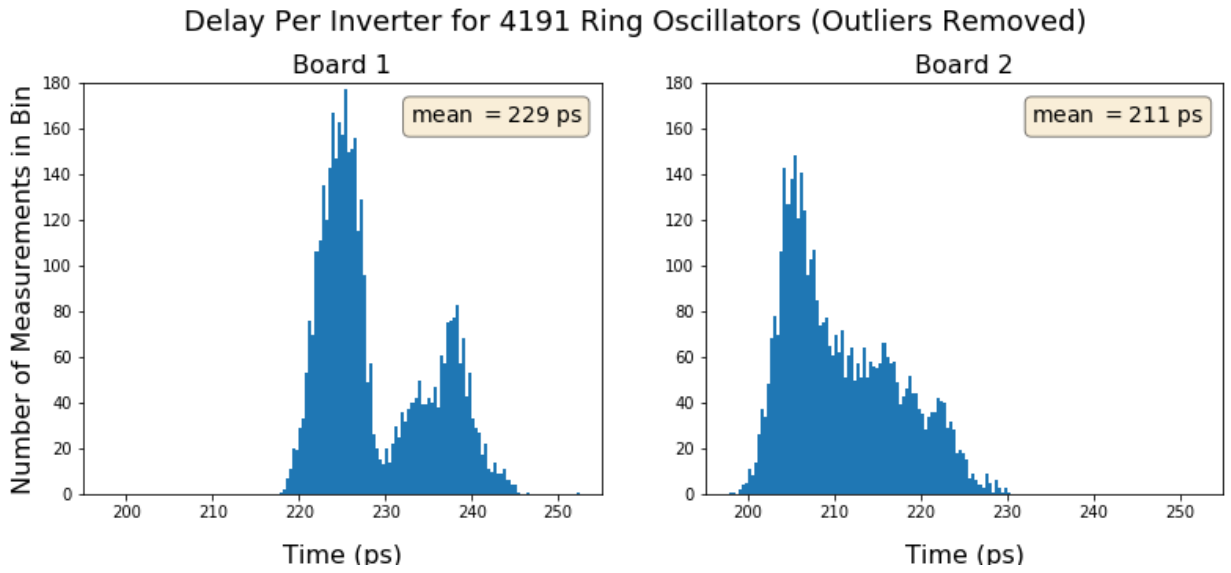


Figure 3.9: Distribution of inverter propagation delays for two different Cyclone V Chips

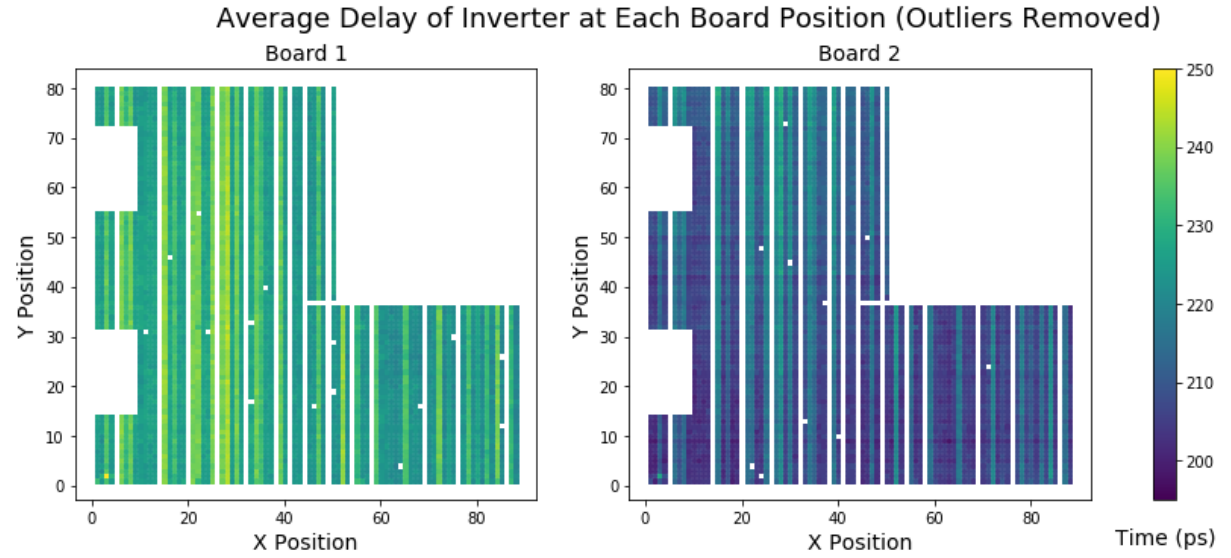


Figure 3.10: Heatmap representation of data in Figure 3.9 where each datapoint is the propagation delay of an inverter in a LAB or MLAB at that coordinate on the chip

3.5 Ring Oscillators with Specified Routing

By checking the layout of many ring oscillators in the *Quartus* **Resource Property Editor**, it became clear that a variety of inputs to the LUT are used to route the design. In addition, it appears as though the *Quartus* Fitter implements the ring oscillators within the same column of LABs or MLABs with identical routing. Given the homogeneity of the delays along the columns, we became interested in the proposition of fixing the routing within the ring oscillators to make the look-up table routing homogeneous. This was done in the method of Chapter 2.

We were first interested in using routing that could be implemented across the entire design. However, in the “normal" mode ALM with one inverter, it seemed as though only DATAE and DATAF are possible inputs to an inverter. Therefore, each ring oscillator had one input of DATAF in the 19th inverter. We then implemented three different designs with the rest of the 18 inverters set to DATAF, DATAC, and DATAD, respectively.

In Figure 3.11, we can see the initial results of this inquiry. It appears that a ring oscillator using input DATAF has very similar delays to the ring oscillators routed by Quartus. However, the ring oscillators that use inputs DATAC and DATAD had a delay per inverter that is almost twice as high as the Quartus defined ring oscillator.

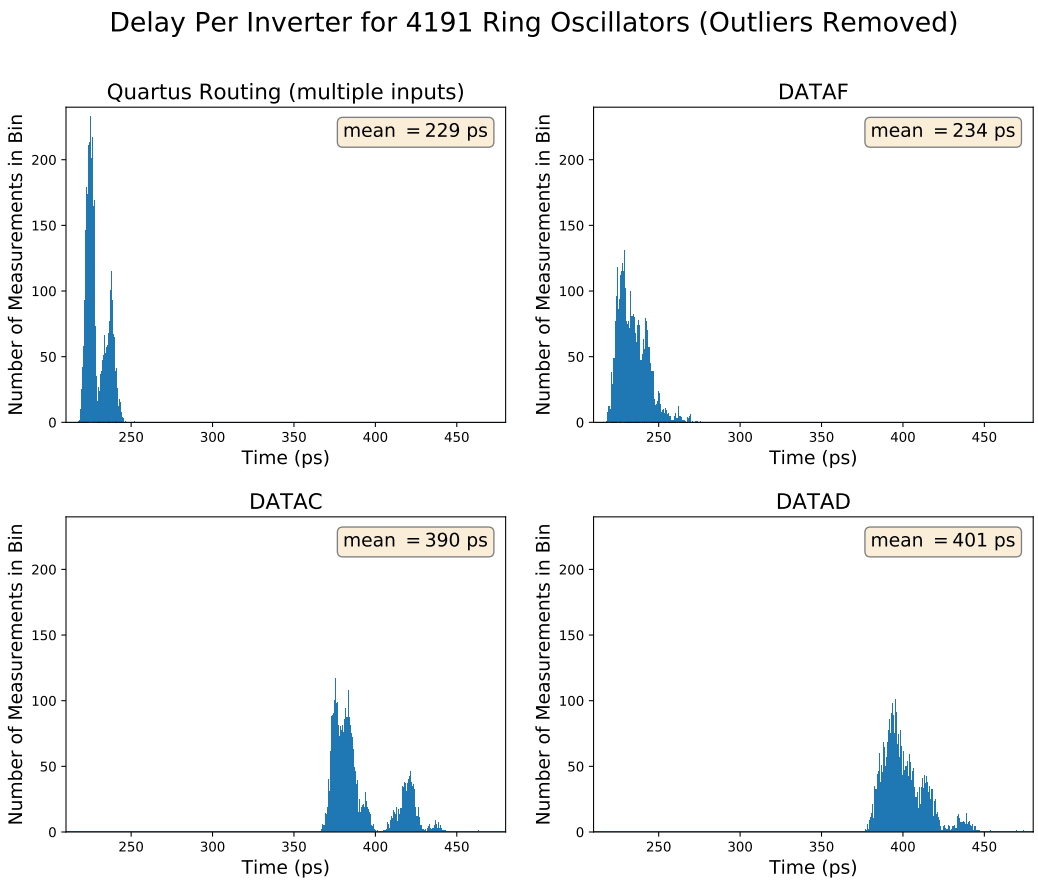


Figure 3.11: Histograms of the average delay per inverter for 19 stage ring oscillators with four different types of routing through the LUT. The four types are *Quartus* defined, DATAF, DATAC, and DATAD.

In Figure 3.12, stacked histograms separated by LAB vs. MLAB are shown for the measurement with the original routing generated by *Quartus* and the measurement of the ring oscillators using input DATAF. When DATAF is used as the LUT input, the distribution becomes much wider and the two peaks are less distinguishable. The delay per inverter of the MLABs has a mean of 245 ps which is slower than the delay per inverter of the LABs with a mean of 231 ps. Moreover, there is a small set of ring oscillators with a delay per inverter above 250 ps. Nearly all of those ring oscillators were in MLABs.

Figure 3.13 is a heatmap of the inverter delay with respect to the coordinates of the LAB/MLAB on the chip. Noting that the MLABs are still slower than the LABs, the delays appear to remain extremely homogenous within the columns themselves. This indicates that the routing within the LABs and MLABs may not be the cause of the homogenous delay within columns. Instead, this could be due to how the local and global interconnect routing is constrained by the structure and fabrication of the chip. As MLABs are also configurable as memory blocks, there may be additional sources of delay.

In Figure 3.14, stacked histograms separated by LAB vs MLAB are shown for the measurements with inputs DATAC and DATAD. The distribution for DATAC not only retained two peaks, but in fact those peaks became more distinguished with very little overlap. The faster peak has a mean of 381 ps and is composed of almost entirely LABs while the slower peak has a mean of 420 ps and is composed of almost entirely MLABs. On the other hand, the distribution for DATAD exhibited the opposite behavior, with the peaks become less distinguished. Recall that this also occurred when DATAF was used as the input. The delay per inverter within the LABs for DATAD was on average slower than for DATA C at 396 ps vs 381 ps. However, the MLABs which used DATAD as an input were actually faster with a mean of 416 ps vs 420 ps for DATAC. Like DATAF, both histograms also featured a small tail of MLABs that oscillated at slower frequencies with a delay per inverter of 430-445 ps for DATAC and 415-445 ps for DATAD. Figure 3.15 is a heatmap of the inverter delay with respect to the coordinates of the LAB/MLAB on the chip for the measurements of DATAC and DATAD ring oscillators. Visually, there is less variation within columns. This plot also reveals that there were a large number of third harmonic outliers in the measurement using input D.

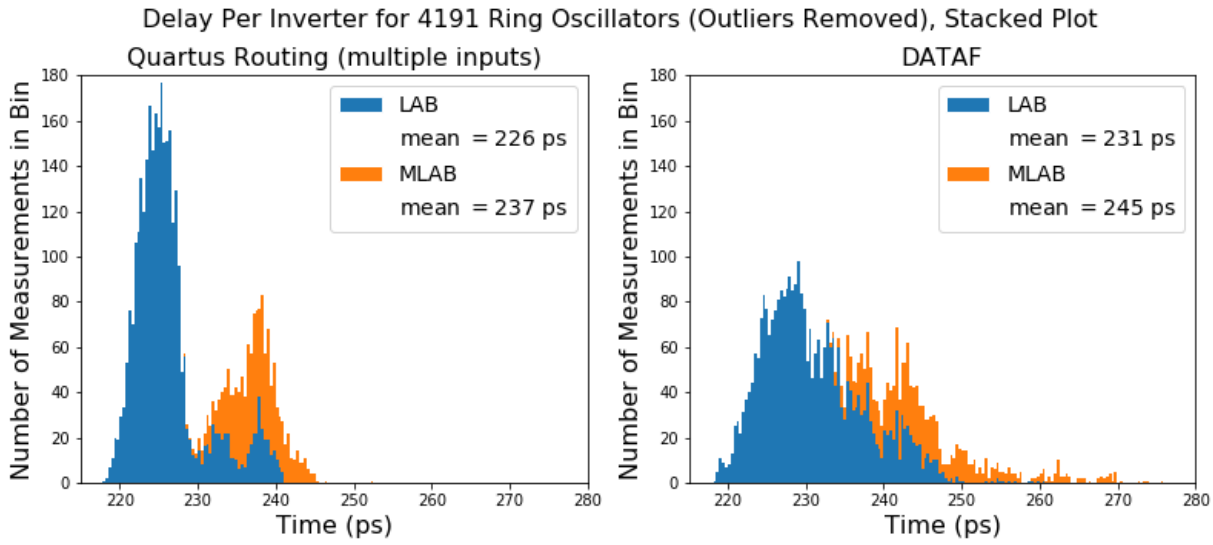


Figure 3.12: Stacked histograms of the average delay per inverter for 19 stage ring oscillators with two different types of routing through the LUT. The two types are *Quartus* defined, and DATAF. The blue represents measurements from the ring oscillators in LABs and the orange represents measurements from the ring oscillators in MLABs.

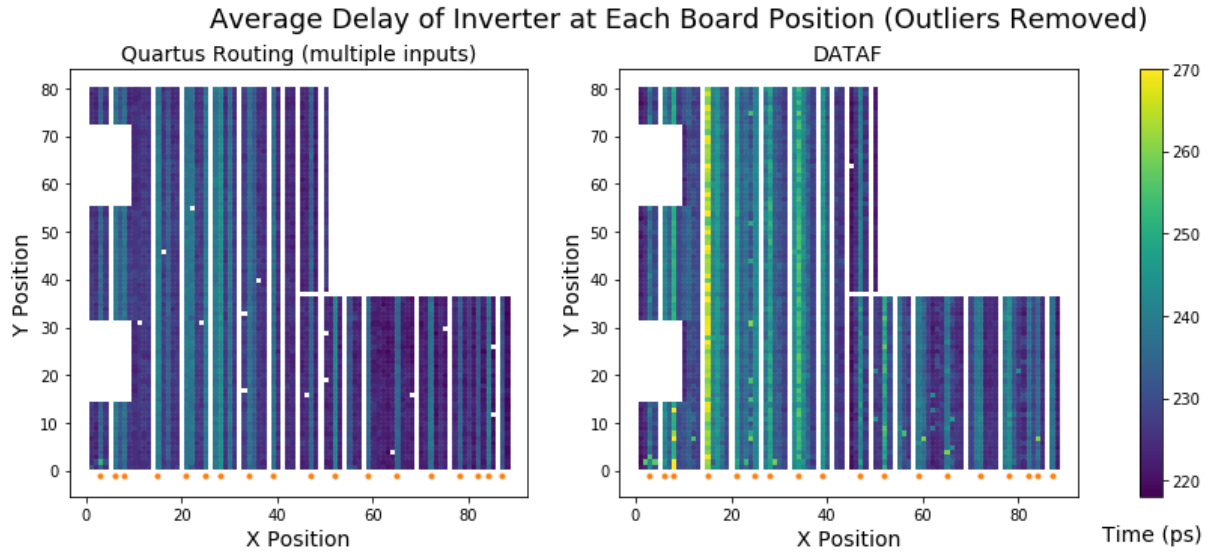


Figure 3.13: Heatmaps of the delay per inverter given the coordinate of the LAB/MLAB. The plot on the left shows the measurement with original routing while the plot on the right shows the measurement with input DATAF. The orange dots denote the columns of MLABs.

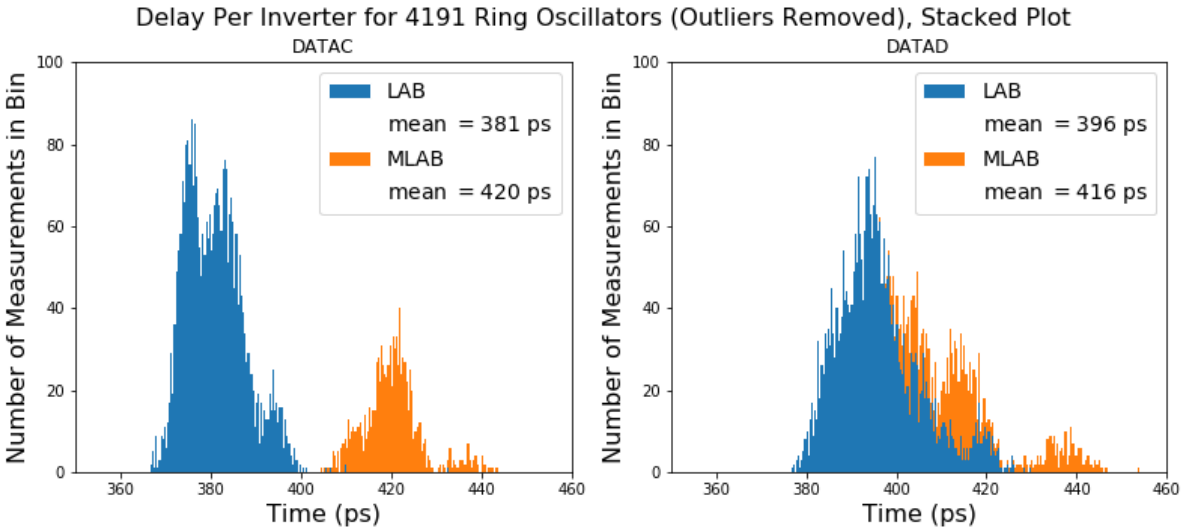


Figure 3.14: Stacked histograms of the average delay per inverter for 19 stage ring oscillators with two different types of routing through the LUT. The left uses DATAC as an input and the right uses DATAD as an intput. The blue represents measurements from the ring oscillators in LABs and the orange represents measurements from the ring oscillators in MLABs.

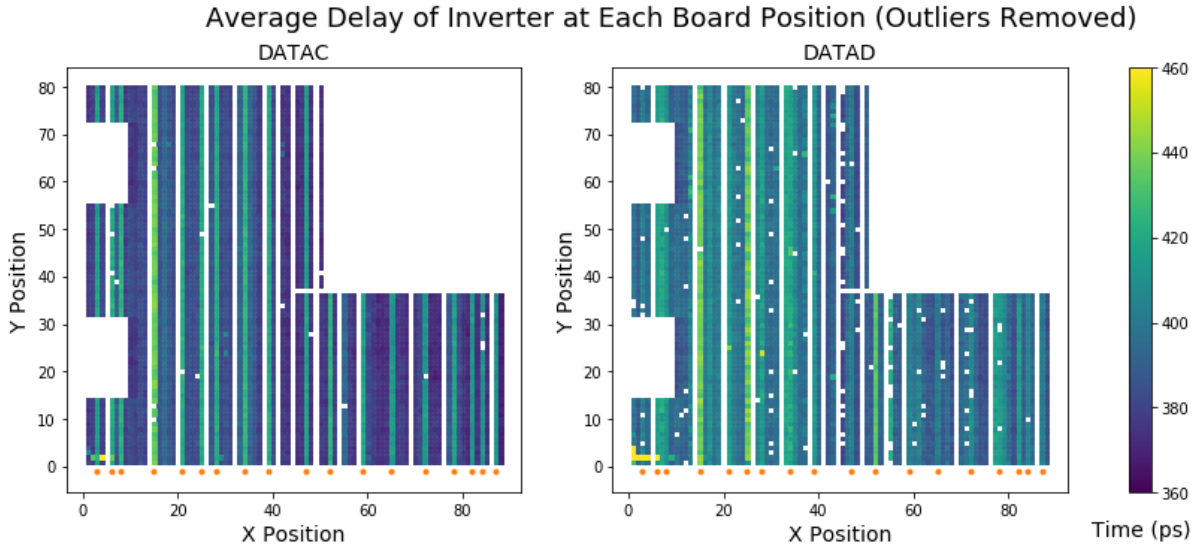


Figure 3.15: Heatmaps of the delay per inverter given the coordinate of the LAB/MLAB. The plot on the left shows the measurement with DATAC and the plot on the right shows the measurement with input DATAD. The orange dots denote the columns of MLABs.

The structure the Cyclone V ALM provides some insight about the inverter delay for inputs C and D. A diagram of the ALM can be found in the Cyclone V Device handbook [3]. A more general schematic of a look-up table is given in *Hardware Security and Trust* [9]. To implement the inverter using input port C or D, more operations are performed on the input signal, potentially leading to a larger delay.

3.6 ROs of Different stages

We also considered the delay per inverter in ring oscillators with various numbers of stages. Ring oscillators of different numbers of stages were implemented within the same LAB. These ring oscillators were allowed to run for 10 seconds, and then measured for one millisecond with ten milliseconds in between measurements. The delay per inverter was computed from these measurements and then used to find the average delay per inverter. This is shown in Figure 3.16. There does not seem to be a clear correlation between the number of stages and the delay of an inverter in a ring oscillator. Moreover, the measurements are not within the error of one another. As frequency of a ring oscillator was shown to decrease after running for an extended period of time in Figure 3.1, the measurements were not run consecutively. However, this led to the possibility of environmental changes. As the ring oscillator frequency is sensitive to environmental changes [15], this could have lead to changes in the measurements that dominated over small differences in ring oscillators with different number of stages. Therefore, it is preferable to perform this measurement repeatedly for each number of stages in as controlled an environment as possible. In addition, it could be beneficial to repeat this measurement in a variety of board locations.

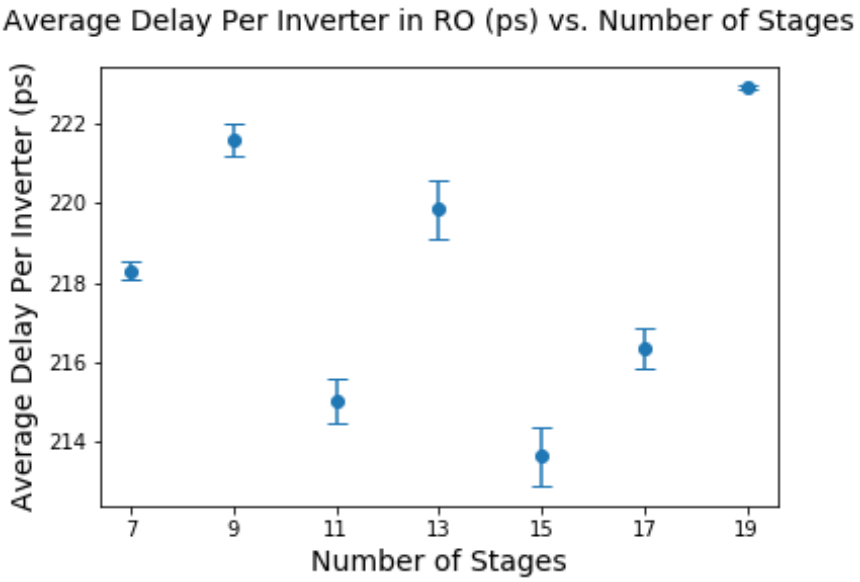


Figure 3.16: The average delay per inverter for ring oscillators with various numbers of stages. Each ring oscillator was measured 600 times for one millisecond with measurements seperated by 10 milliseconds. For each measurement, the delay per inverter was calculated. Then, the average of of those delays was computed for each number of stages. The errorbars are standard deviation.

Chapter 4: Contributions and Future Work

FPGAs can be effective platforms for circuits with a time measurement component such as time-to-digital converters or delay based physically unclonable functions. However, we have found that many factors can significantly impact the delay of an inverter on the FPGA. These includes board location, type of resource (LAB vs MLAB), logic cell routing, and thermalization. Moreover, our results suggest that certain parts of the chip have more similar delays. Understanding this information can allow a TDC or PUF designer to make the most consistent reliable electronics. The process implemented in this thesis can be applied to a variety of Intel FPGAs, especially those which utilize the ALM structure.

There are many ways the results of this thesis can be improved upon and extended

- Repeating the measurement of the ring oscillator over time for a longer time period and with more trials
- Characterizing the board after full thermalization and stabilization
- Investigating the structure of the Adaptive Logic Module to understand why certain logic cell inputs lead to slower delays
- Studying the contribution of the routing interconnects
- Repeating this study for carry-chains, another popular logic element used for delays due to picosecond latency

- Further study ring oscillators of various stages

Appendix A: Appendix code

A.1 Verilog

A.1.1 Ring Oscillator

This ring oscillator code is adapted from David Rosin's thesis on Autonomous Boolean Networks [13].

```
1  module ring_osc (
2          output s_out     //output of RO sent to counter
3          );
4
5  parameter n = 19;       //odd number of inverters
6
7  //make delay line of inverters
8  wire [n-1:0] delay; /* synthesis keep */
9
10 //connect end to the beginning
11 assign delay[0] = ~delay[n-1];
12
13 //connect output to end of oscillator
14 assign s_out = delay[n-1];
15
16 //initialize loop variable
17 genvar i;
18
19 //implement inverters through 0th and 17th element
20 generate
21         for (i=0; i < (n-1) ; i=i+1)
22         begin: generate_delay
23                 assign delay[i+1] = ~delay[i];
24         end
25 endgenerate
26
27 endmodule
```

A.1.2 Control circuitry

```
module multRO (input clk //50 Mhz clock);

//START_REPLACE
parameter num_RO = 226;         //number of RO placed
//STOP_REPLACE


//1: DECLARE CLOCKS
wire clk_count;             //count for 1 slow clock period (1 ms)

//send clock wires through pll
pll pll_inst (.refclk(clk), .outclk_2(clk_count));


//2: INSTANTIATE MEMORY MODULE

//declare registers and wires for memory
wire clock_sig;
wire wren_sig;                          //wire assigned to wren_reg to pass to RAM
    ↪module
reg wren_reg;                           //register to hold value of wren wire (
    ↪enables mem reading)
reg [19:0] data_sig;            //register for data to be written to memory
reg [15:0] address_sig;         //register for address corresponding to data

assign clock_sig = clk; //assign clock signal to normal clock

//give them some starting values
initial begin
        address_sig <= {16{1'b1}};      //make address_sig maximimum to loop value
        wren_reg <= 1'b1;               //initialize enable register to high (always
            ↪ reading)
        data_sig <= 20'b0;                      //initialize the value we read to
            ↪memory as 0
end

//assign the wren wire to the register holding its value
assign wren_sig = wren_reg;

//instantiate ram
ram ram_inst  (
        .address ( address_sig ),
        .clock ( clk ), //latch memory on falling edge of 50 Mhz clock
        .data ( data_sig ),
        .wren ( wren_sig ),
        .q ( q_sig )
```

```
43          );
44
45
46  //3: INSTATIATE RING OSCILLATORS
47  wire [num_RO-1:0] hold; /*synthesis keep*/   //holds the outputs of the ring
        ↪oscillators
48
49  genvar i;
50  generate
51          for (i=0; i< num_RO; i=i+1) begin: generate_RO
52                  ring_osc ROinst (.s_out(hold[i]));
53          end
54  endgenerate
55
56
57  //4: CREATE KEY SIGNALS
58  wire key_start;         //turns on when count is reached in start_delay
59
60  start_delay start_inst (.clk_in(clk), .start(key_start)); //module to count and flip
        ↪ to 1
61
62
63  //5: CONTROL CIRCUITRY
64
65  reg [15:0] RO_reg;                      //holds which RO you're saving
66
67  //instantiate signals
68  reg endcount;                           //tells to end counting
69  reg read;                                       //tells to read
70  reg save;                                       //tells to save
71
72  initial begin
73          RO_reg <= 16'b0;    //first RO is 0th in array
74          save <= 1'b0;           //initialize save to 0
75          read <= 1'b0;           //initialize read to 0
76          endcount <=1'b0;
77  end
78
79  //increment index at positive edge of control clock
80  always @(posedge clk_count) begin
81          //starts saving when reading done
82          if (read) begin
83                  read <= 1'b0;
84                  save <= 1'b1;
85          end
86          //starts clearing when saving done
87          else if (save && (RO_reg >= num_RO)) begin
```

```
88                 save <= 1'b0;
89                 endcount <= 1'b1;
90         end
91         //starts counting if all signals off
92         else if (key_start && !read && !save && !endcount) begin
93                 read <= 1'b1;
94         end
95 end
96
97
98 //6: DO COUNTING
99
100 reg [19:0] counter [num_RO-1: 0];  /*synthesis preserve*/
101
102 //initialize values of the counter
103 genvar count_i;
104 generate
105 for (count_i = 0; count_i < num_RO; count_i= count_i+1) begin: counter_init
106         initial begin
107                 counter[count_i] <= 20'b0;
108         end
109 end
110 endgenerate
111
112 //start counting for each RO
113
114 genvar RO_n;
115 generate
116 for (RO_n = 0; RO_n < num_RO; RO_n = RO_n+1) begin: gen_count
117         always @(posedge hold[RO_n]) begin
118                 // read in on pos edge and increment address
119                 if (read) begin
120                         counter[RO_n] <= counter[RO_n]+1;
121                 end
122         end
123 end
124 endgenerate
125
126
127 //7: SAVE TO MEMORY
128
129 reg ordering;   //ordering variable makes address and data updated on alternating
    ↪cycle
130 initial begin
131         ordering <= 1'b0;
132 end
133
```

```
always @(posedge clk) begin
        if (save && (RO_reg < num_RO) && !ordering) begin
                address_sig <= address_sig + 16'b1;
                ordering <= 1'b1;
        end
        else if (save && (RO_reg < num_RO) && ordering) begin
                data_sig <= counter[address_sig];
                ordering <= 1'b0;
                RO_reg <= RO_reg + 16'b1;
        end

end
endmodule
```

A.2 Python

A.2.1 Compiling

compile_quartus (Python function)

This function compiles the project in four steps (Analysis and Synthesis, Fitter, Assembler, Timing Analyzer). It uses the os library in Python to execute tcl commands to perform each step.

Inputs:

- **project_path**: path to *Quartus* top-level working directory w.r.t. parent directory of the project
- **project_name**: the *Quartus* project name

Outputs: None

```
def compile_quartus(project_path, project_name):
    project = project_path+ project_name
    os.system('quartus_map '+project)
    os.system('quartus_fit '+project)
    os.system('quartus_asm '+project)
    os.system('quartus_sta '+project)
```

A.2.2 Location assignments

Sample location assignments

```
set_location_assignment LABCELL_X13_Y13_N0 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[0]"
set_location_assignment LABCELL_X13_Y13_N3 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[1]"
set_location_assignment LABCELL_X13_Y13_N6 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[2]"
set_location_assignment LABCELL_X13_Y13_N9 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[3]"
set_location_assignment LABCELL_X13_Y13_N12 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[4]"
set_location_assignment LABCELL_X13_Y13_N15 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[5]"
set_location_assignment LABCELL_X13_Y13_N18 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[6]"
set_location_assignment LABCELL_X13_Y13_N21 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[7]"
set_location_assignment LABCELL_X13_Y13_N24 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[8]"
set_location_assignment LABCELL_X13_Y13_N27 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[9]"
set_location_assignment LABCELL_X13_Y13_N30 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[10]"
set_location_assignment LABCELL_X13_Y13_N33 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[11]"
set_location_assignment LABCELL_X13_Y13_N36 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[12]"
set_location_assignment LABCELL_X13_Y13_N39 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[13]"
set_location_assignment LABCELL_X13_Y13_N42 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[14]"
set_location_assignment LABCELL_X13_Y13_N45 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[15]"
set_location_assignment LABCELL_X13_Y13_N48 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[16]"
set_location_assignment LABCELL_X13_Y13_N51 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[17]"
set_location_assignment LABCELL_X13_Y13_N54 -to "ring_osc:generate_RO[27].ROinst|
    ↪delay[18]"

set_location_assignment LABCELL_X13_Y12_N0 -to Add27~1
set_location_assignment LABCELL_X13_Y12_N3 -to Add27~5
set_location_assignment LABCELL_X13_Y12_N6 -to Add27~9
set_location_assignment LABCELL_X13_Y12_N9 -to Add27~13
set_location_assignment LABCELL_X13_Y12_N12 -to Add27~17
set_location_assignment LABCELL_X13_Y12_N15 -to Add27~21
set_location_assignment LABCELL_X13_Y12_N18 -to Add27~25
set_location_assignment LABCELL_X13_Y12_N21 -to Add27~29
```

```
set_location_assignment LABCELL_X13_Y12_N24 -to Add27~33
set_location_assignment LABCELL_X13_Y12_N27 -to Add27~37
set_location_assignment LABCELL_X13_Y12_N30 -to Add27~41
set_location_assignment LABCELL_X13_Y12_N33 -to Add27~45
set_location_assignment LABCELL_X13_Y12_N36 -to Add27~49
set_location_assignment LABCELL_X13_Y12_N39 -to Add27~53
set_location_assignment LABCELL_X13_Y12_N42 -to Add27~57
set_location_assignment LABCELL_X13_Y12_N45 -to Add27~61
set_location_assignment LABCELL_X13_Y12_N48 -to Add27~65
set_location_assignment LABCELL_X13_Y12_N51 -to Add27~69
set_location_assignment LABCELL_X13_Y12_N54 -to Add27~73
set_location_assignment FF_X13_Y12_N2 -to counter[27][0]
set_location_assignment FF_X13_Y12_N5 -to counter[27][1]
set_location_assignment FF_X13_Y12_N8 -to counter[27][2]
set_location_assignment FF_X13_Y12_N11 -to counter[27][3]
set_location_assignment FF_X13_Y12_N14 -to counter[27][4]
set_location_assignment FF_X13_Y12_N17 -to counter[27][5]
set_location_assignment FF_X13_Y12_N20 -to counter[27][6]
set_location_assignment FF_X13_Y12_N23 -to counter[27][7]
set_location_assignment FF_X13_Y12_N26 -to counter[27][8]
set_location_assignment FF_X13_Y12_N29 -to counter[27][9]
set_location_assignment FF_X13_Y12_N32 -to counter[27][10]
set_location_assignment FF_X13_Y12_N35 -to counter[27][11]
set_location_assignment FF_X13_Y12_N38 -to counter[27][12]
set_location_assignment FF_X13_Y12_N41 -to counter[27][13]
set_location_assignment FF_X13_Y12_N44 -to counter[27][14]
set_location_assignment FF_X13_Y12_N47 -to counter[27][15]
set_location_assignment FF_X13_Y12_N50 -to counter[27][16]
set_location_assignment FF_X13_Y12_N53 -to counter[27][17]
set_location_assignment FF_X13_Y12_N56 -to counter[27][18]
set_location_assignment FF_X13_Y12_N59 -to counter[27][19]
```

generate_grid (Python function)

This Python function indicates which x-y positions are increments of 5 apart and 4 apart relative to starting coordinates min_x and min_y and LABs or MLABS, and inputs this information into 2D arrays. These 2D arrays represent the grid of LABs and MLABS which are then placed with a ring oscillator. This uses Noelo's devices.py dictionary which contains information about the type of resource at each coordinate on the board.

Inputs:

- **lab_grid**: passed by reference, lab_grid[x,y] = 1 if a RO will be placed there, 0 if not

- **min_x**: starting x coordinate to position grid
- **min_y**: starting y coordinate to position grid

Outputs: None

```
def generate_grid(lab_grid, mlab_grid,min_x,min_y):
    ##use devices.py file to check the type of each coordinate I want to place RO,
        ↪if LAB or MLAB place 1 in one of grids
    for i in range (min_x, (max_x+1), x_step):
        if (device[(i,min_y)]["TYPE"] == "LAB"):
            for j in range(min_y, max_y+1, y_step):
                if (device[(i,j)]["TYPE"] == "LAB"):
                    lab_grid[(i-1,j-1)] = 1
        elif (device[(i,min_y)]["TYPE"]=="MLAB"):
            for j in range(min_y, max_y+1, y_step):
                if (device[(i,j)]["TYPE"] == "MLAB"):
                    mlab_grid[(i-1,j-1)] = 1
    return()
```

write_verilog (Python function)

This Python function writes the number of ring oscillators placed into main Verilog file of the project, proj_name + '.v'. This number is found in the Python function qsf_assignments. This is so that the correct number of ring oscillators is instantiated into the project, as different grids correspond to different numbers of ring oscillators. It does this by parsing the Verilog file for key comments and rewriting that single line with the new number.

Inputs:

- **path_name**: path to directory with qsf assignments, typically the same as top level *Quartus* project directory
- **proj_name**: *Quartus* project name
- **num_RO**: total number of ring oscillators to be instatiated in the Verilog

Outputs: None

```
def write_verilog(path_name,proj_name,num_RO):
    ver_path = path_name + proj_name + '.v'

    ver = open(ver_path, 'r')
    ver_lines = ver.readlines()
    ver.close()
```

```
7
8     new_ver_lines = []
9     parameter_line = 'parameter num_RO = '+ str(int(num_RO)) +'; \t//number of RO
          ↪placed \n'
10    '''IF YOU ADD STOP REPLACE AND START REPLACE IT IS CONVENIENT TO REPLACE YOUR
          ↪PARAMETER LINES'''
11    read_sig = 1
12    for i in ver_lines:
13        if (i.find('//STOP_REPLACE') != -1):
14            read_sig = 1
15        if (read_sig == 1):
16            new_ver_lines.append(i)
17        if (i.find('//START_REPLACE') != -1):
18            read_sig = 0
19            new_ver_lines.append(parameter_line)
20
21    new_ver = open(ver_path,'w')
22
23    for i in new_ver_lines:
24        new_ver.write(i)
25    new_ver.close()
26
27    return()
```

qsf_assignments (Python function)

This Python function uses functions lab_grid and mlab_grid to write location assignments for qsf file, which differ if it is a LAB or MLAB. This function calls the functions place_lab and place_mlab, which contain the many lines of text required to place a single LAB or MLAB, and to additionally place its counter above or below it. It does this by iterating from 1 to 88 in y-position and 1 to 80 in x-position and checking if either lab_grid[x,y] = 1 or mlab_grid[x,y]= 1. If lab_grid[10,12] = 1, for example, it will call the function place_lab(10,12,4,RO_assign)

Inputs:

- **lab_grid**: lab_grid[x,y] = 1 if a RO will be placed there, 0 if not
- **mlab_grid**: mlab_grid[x,y] = 1 if a RO will be placed there, 0 if not

Outputs:

- **RO_assign**: holds all lines of text to write to qsf file, is passed into place_lab and place_mlab
- **RO_ref**: holds lines of text for reference file, indication the index of the RO and its X-Y location
- **index_RO**: holds the index of the RO while running, and at the end is equal to the number of ROs placed

```
def generate_grid(lab_grid, mlab_grid,min_x,min_y):
    ##use devices.py file to check the type of each coordinate I want to place RO,
        ↪if LAB or MLAB place 1 in one of grids
    for i in range (min_x, (max_x+1), x_step):
        if (device[(i,min_y)]["TYPE"] == "LAB"):
            for j in range(min_y, max_y+1, y_step):
                if (device[(i,j)]["TYPE"] == "LAB"):
                    lab_grid[(i-1,j-1)] = 1
        elif (device[(i,min_y)]["TYPE"]=="MLAB"):
            for j in range(min_y, max_y+1, y_step):
                if (device[(i,j)]["TYPE"] == "MLAB"):
                    mlab_grid[(i-1,j-1)] = 1
    return()
```

place_lab, place_mlab

Writes location assignments for each inverter in a ring oscillator in a LAB at coordinate x_loc, y_loc. Calls on global variables, and start_inv and num_inv to determine the number of inverters that must be placed as well as where the first inverter is placed. A placement must be made for each individual inverter. Also checks if the coordinates above are a LAB and if not, whether the coordinates below are (there is no LAB without a LAB at least above or below it). It then places the counter for that ring oscillator in the LAB above or below by placing the individual adders and registers. RO_assign is passed by reference, and the lines are written into the vector one by one. Called by qsf_assignments function.

Inputs:

- **x_loc**: x coordinate of the LAB for the RO
- **y_loc**: y coordinate of the LAB for the RO
- **index_RO**: the index of the RO you are placing (used to identify it to the project, as they were instantiated in a for loop in Verilog)

- **RO_assign**: holds location assignments to write to qsf in write_qsf

Outputs: None

Description of place_mlab is essentially identically. Source code for both functions is provided below

```
def place_lab(x_loc,y_loc,index_RO,RO_assign):

    for k in range (0,num_inv,1):
        loc_ass = 'set_location_assignment␣LABCELL_X'+str(x_loc) + '_Y'+str(y_loc)+'
            ↪_N'+str(int(3*(start_inv-1)+ 3*k))+ '␣-to␣"ring_osc:generate_RO[' +
            ↪str(index_RO)+'].ROinst|delay['+str(k)+ ']"' + '\n'
        RO_assign.append(loc_ass)
        loc_ass = ''
    RO_assign.append('\n')

    if (device[(x_loc,y_loc-1)]["TYPE"] == "LAB"):
        for k in range(0, 19, 1):
            loc_ass = 'set_location_assignment␣LABCELL_X'+str(x_loc) + '_Y'+str(
                ↪y_loc-1)+'_N'+str(int(3*k)) + '␣-to␣Add' + str(index_RO) + '~' +
                ↪str(4*k+1) + '\n'
            RO_assign.append(loc_ass)
            loc_ass = ''
        for k in range(0,20,1):
            ff_ass = 'set_location_assignment␣FF_X' + str(x_loc) + '_Y' + str(y_loc
                ↪-1) + '_N' + str(int(3*k+2)) + '␣-to␣counter[' + str(index_RO) +'
                ↪][' + str(int(k))+']' + '\n'
            RO_assign.append(ff_ass)
            ff_ass = ''
    elif(device[(x_loc,y_loc+1)]["TYPE"] == "LAB"):
        for k in range(0, 19, 1):
            loc_ass = 'set_location_assignment␣LABCELL_X'+str(x_loc) + '_Y'+str(
                ↪y_loc+1)+'_N'+str(int(3*k)) + '␣-to␣Add' + str(index_RO) + '~' +
                ↪str(4*k+1) + '\n'
            RO_assign.append(loc_ass)
            loc_ass = ''
        for k in range(0,20,1):
            ff_ass = 'set_location_assignment␣FF_X' + str(x_loc) + '_Y' + str(y_loc
                ↪+1) + '_N' + str(int(3*k+2)) + '␣-to␣counter[' + str(index_RO) +'
                ↪][' + str(int(k))+']' + '\n'
            RO_assign.append(ff_ass)
            ff_ass = ''
    RO_assign.append('\n')
    return()
```

```
def place_mlab(x_loc,y_loc,index_RO,RO_assign):

```

```
    for k in range (0,num_inv,1):
        loc_ass = 'set_location_assignment MLABCELL_X'+str(x_loc) + '_Y'+str(y_loc)
            ↪+'_N'+str(int(3*(start_inv-1)+ 3*k))+ ' -to "ring_osc:generate_RO[' +
            ↪str(index_RO)+'].ROinst|delay['+str(k)+ ']"' + '\n'
        RO_assign.append(loc_ass)
        loc_ass = ''
    RO_assign.append('\n')

    if (device[(x_loc,y_loc-1)]["TYPE"] == "MLAB"):
        for k in range(0, 19, 1):
            loc_ass = 'set_location_assignment MLABCELL_X'+str(x_loc) + '_Y'+str(
                ↪y_loc-1)+'_N'+str(int(3*k)) + ' -to Add' + str(index_RO) + '~' +
                ↪str(4*k+1) + '\n'
            RO_assign.append(loc_ass)
            loc_ass = ''
        for k in range(0,20,1):
            ff_ass = 'set_location_assignment FF_X' + str(x_loc) + '_Y' + str(y_loc
                ↪-1) + '_N' + str(int(3*k+2)) + ' -to counter[' + str(index_RO)
                ↪+'][' + str(int(k))+']' + '\n'
            RO_assign.append(ff_ass)
            ff_ass = ''
    elif(device[(x_loc,y_loc+1)]["TYPE"] == "MLAB"):
        for k in range(0, 19, 1):
            loc_ass = 'set_location_assignment MLABCELL_X'+str(x_loc) + '_Y'+str(
                ↪y_loc+1)+'_N'+str(int(3*k)) + ' -to Add' + str(index_RO) + '~' +
                ↪str(4*k+1) + '\n'
            RO_assign.append(loc_ass)
            loc_ass = ''
        for k in range(0, 20, 1):
            ff_ass = 'set_location_assignment FF_X' + str(x_loc) + '_Y' + str(y_loc
                ↪+1) + '_N' + str(int(3*k+2)) + ' -to counter[' + str(index_RO)
                ↪+'][' + str(int(k))+']' + '\n'
            RO_assign.append(ff_ass)
            ff_ass = ''

    return()
```

write_qsf (Python function)

This Python function takes the output of qsf_assignments, RO_assign and writes it line by line to a file proj_name+ "_source.qsf", which is sourced by the main project .qsf file. This also deletes the main project .qsf file and rewrites it from a template in the parent directory. This is because generating an .rcf file through back-annotating adds location

assignments to the .qsf thereby creating duplicate location assignments for the same index RO.

Inputs:

- **path_name**: path to directory with qsf assignments, typically the same as top level *Quartus* project directory.
- **proj_name**: *Quartus* project name
- **loc_ass**: vector containing lines to write to _source.qsf file

Outputs: None

Code for when back annotate is used

```
def write_qsf(path_name,proj_name,loc_ass):

    ##open main qsf file and template file and rewrite with simple file to remove
        ↪back_annotate assignments
    actual_qsf_path = path_name+proj_name+'.qsf'
    actual_qsf = open(actual_qsf_path,'w')

    template_path = 'kosher_qsf_template.txt'
    template_open = open(template_path,'r')
    template = template_open.readlines()
    template_open.close()

    ##writes lines from template into qsf
    for lines in template:
        actual_qsf.write(lines)

    ##write location assignments into second qsf file
    qsf_path = path_name + proj_name +'_source.qsf'
    qsf = open(qsf_path,'w')

    for i in loc_ass:
        qsf.write(i)

    actual_qsf.close()
    qsf.close()

    return()
```

Code for when back annotate is not used:

```
def write_qsf(path_name,proj_name,loc_ass):
    qsf_path = path_name + proj_name +'_source.qsf'

```

```
    qsf = open(qsf_path,'w')
    for i in loc_ass:
        qsf.write(i)

    qsf.close()

    return()
```

write_ref (Python function)

Writes the vector RO_ref containing lines listing ring oscillator index, x-position, and y-position for referencing when collecting data. Allows this information to be retained without the compile script. run+number is used to name the file, as there is a different reference file for each placement of the ROs These lines are generated in the function qsf_assignments

Inputs:

- **run_number**: which number .sof file it corresponds to
- **RO_ref**: lines to write into reference file containing RO index, x-coordinate, y-coordinate

Outputs: None

```
def write_ref(run_number,RO_ref):
    ref_name = loc_ref_base + str(int(run_number)) + '.txt'
    r_out = open(ref_name,'w')
    for i in RO_ref:
        r_out.write(i)
    r_out.close()
```

A.2.3 Routing Functions

Inverter LUTmask Dictionary

This is a dictionary of LUTmasks corresponding to an inverter given a specific input (i.e. 'DATAA'). The dictionary has the format

```
lutmask_dict[   DATAA   ][#] =    AAAA
```

where # is the index of the LUT (out of the four LUTs in each ALM). This information will be different for FPGAs which do not use the Adaptive Logic Module.

```
## DICTIONARY for Inverter LUT mask of input
## number 0 through 3 represent which LUT MASK

DATAA = {
    0 : 'AAAA', 1 : 'AAAA', 2 : 'AAAA', 3 : 'AAAA'
}

DATAB = {
    0 : 'CCCC', 1 : 'CCCC', 2 : 'CCCC', 3 : 'CCCC'
}

DATAC = {
    0 : 'F0F0', 1 : 'F0F0', 2 : 'F0F0', 3 : 'F0F0'
}

DATAD = {
    0 : 'FF00', 1 : 'FF00', 2 : 'FF00', 3 : 'FF00'
}

##DATAE seems to only happen when it's the only one in a LAB since inversion seems
    ↪to require a specific lut mask that uses all the ALM
DATAE = {
    0 : '0000', 1 : 'FFFF', 2 : '0000', 3 : 'FFFF'
}

DATAF = {
    0 : '0000', 1 : '0000', 2 : 'FFFF', 3 : 'FFFF'
}

lutmask_dict  = {'DATAA' : DATAA, 'DATAB' : DATAB, 'DATAC' : DATAC, 'DATAD' : DATAD,
    ↪ 'DATAE' : DATAE, 'DATAF' : DATAF }

## call using lut_mask_dict['DATAA'][0], etc
```

TCL Template for Adding a Connection

```

##############
# ADD  <INSERT_RO_INDEX>, <INSERT_CURRENT_DELAY>#
##############

set node_properties [ node_properties_record #auto \
        -node_name |multRO|ring_osc:generate_RO\[<INSERT_RO_INDEX>\].ROinst|delay\[<
            ↪INSERT_CURRENT_DELAY>\] \
        -node_type LCCOMB_SII \
        -op_mode <INSERT_MODE> \
        -position <INSERT_POSITION> \
```

```
        -f0_lut_mask <INSERT_LUT0> \
        -f1_lut_mask <INSERT_LUT1> \
        -f2_lut_mask <INSERT_LUT2> \
        -f3_lut_mask <INSERT_LUT3> \
        -fanins [ list \
                [ fanin_record #auto -dst {-port_type <INSERT_OLD_INPUT> -lit_index
                    ↪0} -src {-node_name |multRO|ring_osc:generate_RO\[<
                    ↪INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_PRIOR_DELAY>\]
                    ↪-port_type COMBOUT -lit_index 0} -delay_chain_setting -1 ] \
        ] \
]
set result [ make_ape_connection_wrapper $node_properties |multRO|
    ↪ring_osc:generate_RO\[<INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_CURRENT_DELAY
    ↪>\] <INSERT_NEW_INPUT> 0 |multRO|ring_osc:generate_RO\[<INSERT_RO_INDEX>\]
    ↪.ROinst|delay\[<INSERT_PRIOR_DELAY>\] COMBOUT 0 -1 ]
if { $result == 0 } {
        set had_failure 1
        dump_node $node_properties
}
remove_all_record_instances

###################
```

TCL Template for Removing a Connection

```
#############
# REMOVE  <INSERT_RO_INDEX>, <INSERT_CURRENT_DELAY>#
#############

set node_properties [ node_properties_record #auto \
        -node_name |multRO|ring_osc:generate_RO\[<INSERT_RO_INDEX>\].ROinst|delay\[<
            ↪INSERT_CURRENT_DELAY>\] \
        -node_type LCCOMB_SII \
        -op_mode <INSERT_MODE> \
        -position <INSERT_POSITION> \
        -f0_lut_mask <INSERT_LUT0> \
        -f1_lut_mask <INSERT_LUT1> \
        -f2_lut_mask <INSERT_LUT2> \
        -f3_lut_mask <INSERT_LUT3> \
        -fanins [ list \
                [ fanin_record #auto -dst {-port_type <INSERT_OLD_INPUT> -lit_index
                    ↪0} -src {-node_name |multRO|ring_osc:generate_RO\[<
                    ↪INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_PRIOR_DELAY>\]
                    ↪-port_type COMBOUT -lit_index 0} -delay_chain_setting -1 ] \
```

```
16                 [ fanin_record #auto -dst {-port_type <INSERT_NEW_INPUT> -lit_index
                       ↪0} -src {-node_name |multRO|ring_osc:generate_RO\[<
                       ↪INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_PRIOR_DELAY>\]
                       ↪-port_type COMBOUT -lit_index 0} -delay_chain_setting -1 ] \
17         ] \
18 ]
19 set result [ remove_ape_connection_wrapper $node_properties |multRO|
       ↪ring_osc:generate_RO\[<INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_CURRENT_DELAY
       ↪>\] <INSERT_OLD_INPUT> 0  ]
20 if { $result == 0 } {
21         set had_failure 1
22         dump_node $node_properties
23 }
24 remove_all_record_instances
25
26 ###################
```

TCL Template for Changing a LUTmask

```
1  #############
2  # CHANGELUT  <INSERT_RO_INDEX>, <INSERT_CURRENT_DELAY>#
3  #############
4
5  set node_properties [ node_properties_record #auto \
6          -node_name |multRO|ring_osc:generate_RO\[<INSERT_RO_INDEX>\].ROinst|delay\[<
               ↪INSERT_CURRENT_DELAY>\] \
7          -node_type LCCOMB_SII \
8          -op_mode <INSERT_MODE> \
9          -position <INSERT_POSITION> \
10         -f0_lut_mask <INSERT_LUT0> \
11         -f1_lut_mask <INSERT_LUT1> \
12         -f2_lut_mask <INSERT_LUT2> \
13         -f3_lut_mask <INSERT_LUT3> \
14         -fanins [ list \
15                 [ fanin_record #auto -dst {-port_type <INSERT_NEW_INPUT> -lit_index
                       ↪0} -src {-node_name |multRO|ring_osc:generate_RO\[<
                       ↪INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_PRIOR_DELAY>\]
                       ↪-port_type COMBOUT -lit_index 0} -delay_chain_setting -1 ] \
16         ] \
17 ]
18 set result [ set_lutmask_wrapper $node_properties |multRO|ring_osc:generate_RO\[<
       ↪INSERT_RO_INDEX>\].ROinst|delay\[<INSERT_CURRENT_DELAY>\] "F<INDEX_LUTMASK>␣
       ↪LUT␣Mask" <NEW_LUTMASK> ]
19 if { $result == 0 } {
20         set had_failure 1
21         dump_node $node_properties
22 }
```

```
23 remove_all_record_instances
```

make_rcf_dict

This Python function uses the regular expression library in Python to search for lines of the form

```
1 dest = ( ring_osc:generate_RO[<index_RO>].ROinst|delay[<index_delay>], DATAA ),
    ↪route_port = <original input>;
```

to scrape out <index_RO>, <index_delay> and <original_input> from the routing constraints file and put them in a dictionary where you can use (index_RO,index_delay) to obtain <original_input>

Inputs: None
Outputs:

- **old_input_dict**: A dictionary with entries of the form

```
1 {(index_RO, index_delay) :  DATA #  }
```

where index_RO is the index of the ring oscillator, index_delay is the number of the inverter, and DATA# is which input the LUT originally uses. Can be called by

```
1 old_input_dict[(index_RO,index_delay)] =  DATA #
```

```
1  def make_rcf_dict():
2
3      ##open rcf file and read in lines
4
5      rcf_path = RO_path + RO_proj_name + '.rcf'
6      rcf_file = open(rcf_path, 'r')
7      orig_rcf = rcf_file.readlines()
8      rcf_file.close()
9      old_input_dict = {}
10
11     ##search rcf file for original inputs and insert them into dictionary which
           ↪returns the original input when called by RO_index and inverter_index
12
13     for lines in orig_rcf:
14         check = re.search(r"dest␣=␣\(␣ring_osc:generate_RO\[(?P<RO_index>\w+)\]\.
               ↪ROinst\|delay\[(?P<delay_index>\w+)\],␣DATA(\w+)␣\),␣route_port␣=␣(?P<
               ↪old_input>\w+);", lines)
15         if (check != None):
16             temp_RO_index = int(check.group('RO_index'))
```

```
17
18              temp_delay_index = int(check.group('delay_index'))
19
20              temp_old_input = check.group('old_input')
21
22              old_input_dict.update({(temp_RO_index, temp_delay_index) :
                    ↪temp_old_input})
23
24
25      return(old_input_dict)
```

addremove_connection (Python Function)

This Python function uses one of two tcl templates that were generated using Quartus, "template_add_connection.txt" and "template_remove_connection.txt".

These templates contain keywords such as <INSERT_RO_INDEX> in angled brackets, which are replaced by the correct input for the specific connections, configuration, oscillator, and inverter. A dictionary found in inv_lutmask_dict.py is used to determine each initial LUT mask, which must also be specified in the initial nodes in the tcl script. This takes the initial input value (i.e. DATAA) and returns the LUTmask on each of the four LUT that correspond to inversion of that input. The dictionary has the format lutmask_dict['DATAA'][#] = 'AAAA' where # is the index of the LUT. These inputs are passed to <INSERT_LUT0>, <INSERT_LUT1> etc.

Inputs:

- **template**: lines of a template, currently read in from .txt files with a shell of the .tcl script needed to either add or remove a fanin connection
- **index_RO**: the index of the RO you are changing the routing of, used to define the input signal to the LUT.
  - Replaces keyword <INSERT_RO_INDEX>
- **index_current_delay**: the index of the inverter whose corresponding LUT you are changing the routing of, used to define the input signal to the LUT
  - Replaces keyword <INSERT_CURRENT_DELAY>

- **index_prior_delay**: the index of the inverter whose output signal feeds into the input signal of the current inverter
  - Replaces keyword <INSERT_PRIOR_DELAY>
- **position**: takes on values 'top' or 'bottom', depending on whether it is in the top or bottom of the ALM
  - Replaces keyword <INSERT_POSITION>
- **old_input**: the original input into the LUT
  - Replaces keyword <INSERT_OLD_INPUT>
- **new_input**: the new input into the LUT (in this case DATAF)
  - Replaces keyword <INSERT_NEW_INPUT>
- **mode**: indicates whether two different inverters are being implemented in the ALM (has the value 'fractured') or a single inverter is being implemented in the ALM (has the value 'normal')
  - Replaces keyword <INSERT_MODE>

Outputs:

- tcl_lines: Outputs the lines to write to the tcl script which remove or add the fanin connection

```
def addremove_connection(template,index_RO,index_current_delay,index_prior_delay,
    ↪position,old_input,new_input,mode):

    tcl_lines = []

    ##Replace with dictionary
    ##Could not figure out how to do dictionary replacement now that maketrans is
        ↪depracated
    for lines in template:
        tcl_line = lines.replace('<INSERT_RO_INDEX>',str(index_RO))
        tcl_line = tcl_line.replace('<INSERT_CURRENT_DELAY>',str(index_current_delay
            ↪))
        tcl_line = tcl_line.replace('<INSERT_PRIOR_DELAY>',str(index_prior_delay))
        tcl_line = tcl_line.replace('<INSERT_OLD_INPUT>',old_input)
        tcl_line = tcl_line.replace('<INSERT_NEW_INPUT>',new_input)
        tcl_line = tcl_line.replace('<INSERT_MODE>',mode)
        tcl_line = tcl_line.replace('<INSERT_POSITION>',position)
        tcl_line = tcl_line.replace('<INSERT_LUT0>',lutmask_dict[old_input][0])
        tcl_line = tcl_line.replace('<INSERT_LUT1>',lutmask_dict[old_input][1])
        tcl_line = tcl_line.replace('<INSERT_LUT2>',lutmask_dict[old_input][2])
        tcl_line = tcl_line.replace('<INSERT_LUT3>',lutmask_dict[old_input][3])
```

```
19
20          tcl_lines.append(tcl_line)
21
22      return(tcl_lines)
```

change_lutmask (Python Function)

This Python function uses Quartus generated template "template_change_lutmask.txt". This template contains keywords in angled brackets such as <INSERT_RO_INDEX> and importantly <NEW_LUTMASK> and <INDEX_LUTMASK>, which are replaced by the correct input for the specific connections, configuration, oscillator, and inverter. Unlike the previous script, change_lutmask may call the template multiple times, until all the LUT masks are updated. The mode and position impacts how many and which LUTmasks are editable for a given inverter. Therefore, each initial (configurable) LUTmask is checked with the desired LUTmask to see which must be changed, and the template is called each time. A running list of LUTmask edits are tracked after each call of the template, as the updated LUTmask input must be in the successive initial nodes lists.

Inputs:

- **template**: lines of a template, currently read in from .txt files with a shell of the .tcl script needed to either add or remove a fanin connection
- **index_RO**: the index of the RO you are changing the routing of, used to define the input signal to the LUT.
  - Replaces keyword <INSERT_RO_INDEX>
- **index_current_delay**: the index of the inverter whose corresponding LUT you are changing the routing of, used to define the input signal to the LUT
  - Replaces keyword <INSERT_CURRENT_DELAY>
- **index_prior_delay**: the index of the inverter whose output signal feeds into the input signal of the current inverter
  - Replaces keyword <INSERT_PRIOR_DELAY>
- **position**: takes on values 'top' or 'bottom', depending on whether it is in the top or bottom of the ALM

- Replaces keyword <INSERT_POSITION>

- **old_input**: the original input into the LUT
  - Replaces keyword <INSERT_OLD_INPUT>
- **new_input**: the new input into the LUT (in this case DATAF)
  - Replaces keyword <INSERT_NEW_INPUT>
- **mode**: indicates whether two different inverters are being implemented in the ALM (has the value 'fractured') or a single inverter is being implemented in the ALM (has the value 'normal')
  - Replaces keyword <INSERT_MODE>

Outputs:

- **tcl_lines**: Outputs the lines to write to the tcl script which remove or add the fanin connection

```
def change_lutmask(template,index_RO,index_current_delay,index_prior_delay,position,
    ↪old_input,new_input,mode):
    ##List to hold strings to write to tcl script
    tcl_lines = []

    ##length four list which has a 1 if the lut needs to be changed and a 0 if not
    change_lut = np.zeros(4)

    ##length four list which holds the status of each lutmask & is edited when they
        ↪change
    current_lut_status = [lutmask_dict[old_input][0],lutmask_dict[old_input][1],
        ↪lutmask_dict[old_input][2],lutmask_dict[old_input][3]]


    ##This loop checks which LUT must be changed
    ##if the mode of the ALM is 'normal' all four lutmasks must be configured
        ↪correctly
    if (mode == 'normal'):
        for i in range(4):
            if (lutmask_dict[old_input][i] != lutmask_dict[new_input][i]):
                change_lut[i] = 1
    ##if the mode is fractured, only two must be. These two are dependent on the
        ↪position in the LUT
    elif (position == 'top'):
        if (lutmask_dict[old_input][0] != lutmask_dict[new_input][0]):
                change_lut[0] = 1
        if (lutmask_dict[old_input][2] != lutmask_dict[new_input][2]):
                change_lut[2] = 1
```

```
    elif (position == 'bottom'):
        if (lutmask_dict[old_input][1] != lutmask_dict[new_input][1]):
                change_lut[1] = 1
        if (lutmask_dict[old_input][3] != lutmask_dict[new_input][3]):
                change_lut[3] = 1

    ##FUTURE: Will replace with dictionary
    ##Could not figure out how to do dictionary replacement now that maketrans is
        ↪depracated

    ##This loop checks if the i-th lutmask must be changed and adds lines to the tcl
        ↪ script if it must
    for i in range(4):
        if change_lut[i] == 1:
            for lines in template:
                tcl_line = lines.replace('<INSERT_RO_INDEX>',str(index_RO))
                tcl_line = tcl_line.replace('<INSERT_CURRENT_DELAY>',str(
                    ↪index_current_delay))
                tcl_line = tcl_line.replace('<INSERT_PRIOR_DELAY>',str(
                    ↪index_prior_delay))
                tcl_line = tcl_line.replace('<INSERT_OLD_INPUT>',old_input)
                tcl_line = tcl_line.replace('<INSERT_NEW_INPUT>',new_input)
                tcl_line = tcl_line.replace('<INSERT_MODE>',mode)
                tcl_line = tcl_line.replace('<INSERT_POSITION>',position)

                tcl_line = tcl_line.replace('<INSERT_LUT0>',current_lut_status[0])
                tcl_line = tcl_line.replace('<INSERT_LUT1>',current_lut_status[1])
                tcl_line = tcl_line.replace('<INSERT_LUT2>',current_lut_status[2])
                tcl_line = tcl_line.replace('<INSERT_LUT3>',current_lut_status[3])

                tcl_line = tcl_line.replace('<INDEX_LUTMASK>',str(i))
                tcl_line = tcl_line.replace('<NEW_LUTMASK>',lutmask_dict[new_input][
                    ↪i])

                tcl_lines.append(tcl_line)
            if (mode == 'fractured') and (i < 2):
                current_lut_status[0] = lutmask_dict[new_input][0]
                current_lut_status[1] = lutmask_dict[new_input][1]
            elif (mode == 'fractured') and (i > 1):
                current_lut_status[2] = lutmask_dict[new_input][2]
                current_lut_status[3] = lutmask_dict[new_input][3]
            else:
                current_lut_status[i] = lutmask_dict[new_input][i]


            ##once the tcl script is written, we must update current lutmask status
                ↪to ensure our nodes are correct in the next change
```

```
65
66     return(tcl_lines)
```

Bibliography

[1]

[2]

[3] *Cyclone V Device Handbook Volume 1: Device Interfaces and Integration*. 2020.

[4] I. L. Markov A. B. Kahng, J. Lienig and J. Hu. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer, First edition, 2011. DOI: DOI 10.1007/978-90-481-9591-6.

[5] S. Akthar. Fpga architecture, 2014. Accessed on 2021-04-09.

[6] F. Dadouche et. al. New design-methodology of high-performance tdc on a low cost fpga targets. *Sensors and Transducers*, 193(10):123–134, October 2015.

[7] S. Gandhare and B. Karthikeyan. Survey on fpga architecture and recent applications. In *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, pages 1–4, 2019. DOI: 10.1109/ViTECoN.2019.8899550.

[8] F. Fung J. Chromczack. J. Stephenson, D. Chen. *Understanding Metastability in FPGAs*. 2009.

[9] G. Di Natale N. Sklavos, R. Chaves and F. Regazzoni. *Hardware Security and Trust: Design and Deployment of Integrated Circuits in a Threatened Environment*. Springer, 2017.

[10] S. R. Nassif. Design for variability in dsm technologies. *Proc.IEEE International Symposium on Quality Electronic Design*, pages 451–454, 2000. DOI: 10.1109/ISQED.2000.838919.

[11] S. Ogrenci-Memik. *Heat Management in Integrated Circuits: On-chip and System-level Monitoring and Cooling*. Institution of Engineering and Technology, 2015.

[12] Maes R. *Physically Unclonable Functions: Concept and Constructions. In: Physically Unclonable Functions*. Springer, Berlin, Heidelberg, 2013. DOI: 10.1007/978-3-642-41395-7_2.

[13] D. Rosin. *Dynamics of Complex Autonomous Boolean Networks*. PhD thesis, Technischen Universität Berlin, 2014.

[14] P. Sedcole and P. Y. K. Cheung. Within-die delay variability in 90nm fpgas and beyond. In *2006 IEEE International Conference on Field Programmable Technology*, pages 97–104, 2006. DOI:10.1109/FPT.2006.270300.

[15] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *2007 44th ACM/IEEE Design Automation Conference*, pages 9–14, 2007.

[16] T. Werner and V. Akella. "Asynchronous processor survey". *IEEE Comput. Mag.*, 30(11):67–76, Nov. 1997. DOI: 10.1109/2.634866.

[17] S. F. Al-Sarawi Y. Gao and D. Abbott. Physical unclonable functions. *Nat Electron*, 3:81–91, Feb. 2020. DOI: 10.1038/s41928-020-0372-5.